Hierarchies in classes of program schemes

Argimiro A. Arratia-Quesada[*]
Departamento de Matemáticas, Universidad Simón Bolívar,
Apartado 89000, Caracas 1080-A, Venezuela

Savita R. Chauhan[†]
Department of Computer Science,
University of Wales Swansea, Swansea SA2 8PP, U.K.

Iain A. Stewart[‡]
Department of Mathematics and Computer Science,
University of Leicester, Leicester LE1 7RH, U.K.

Abstract

We begin by proving that the class of problems accepted by the program schemes of NPS is exactly the class of problems defined by the sentences of transitive closure logic (program schemes of NPS are obtained by generalizing basic non-deterministic while-programs whose tests within while instructions are quantifier-free first-order formulae). We then show that our program schemes form a proper infinite hierarchy within NPS whose analogy in transitive closure logic is a proper infinite hierarchy, the union of which is full transitive closure logic but for which every level of the hierarchy has associated with it a first-order definable problem not in that level. We then proceed to add a stack to our program schemes, so obtaining the class of program schemes NPSS, and characterize the class of problems accepted by the program schemes of NPSS as the class of problems defined by the sentences of path system logic. We show that there is a proper infinite hierarchy within NPSS, with an analogous hierarchy within path system logic (again, such that every level of the hierarchy has associated with it a first-order definable problem not in that level). Like the hierarchies in transitive closure logic and NPS, the hierarchies in path system logic and NPSS are all proper even when we consider only problems involving undirected trees or problems involving out-trees. One aspect of our analysis that we believe to be particularly interesting is that we do not use Ehrenfeucht-Fraïssé games for our inexpressibility results, as is usually the case in finite model theory, but we simply consider computations of program schemes on certain finite structures.

[*]Supported by EPSRC Grant GR/L 92549.
[†]Some of the results in this paper will appear in this author's forthcoming Ph.D. thesis.
[‡]Supported by EPSRC Grant GR/K 96564.

1 Introduction

There is a strong relationship between finite model theory and computational complexity theory, the outstanding conduit probably being Fagin's Theorem [15] which equates the complexity class **NP**, the class of problems solvable in non-deterministic polynomial-time, with Σ_1^1, the class of problems definable by the sentences of existential second-order logic. This beautiful and succinct result lies at the root of the tree that has since grown linking finite model theory and computational complexity, and there is a plethora of results detailing logical characterizations of numerous different complexity classes ranging from $\mathbf{AC}_0$, the class of problems accepted by constant-depth polynomial-size Boolean circuits, to **PSPACE**, the class of problems solvable in deterministic polynomial-space, and beyond (the references [1, 4, 12, 20, 28, 29, 33, 37, 41, 42, 44, 45, 49] include a selection of such characterizations).

It is all very well logically capturing complexity classes; but what can one do with these characterizations? For on the face of it, they simply provide translations of (hard) complexity theoretic questions into finite model theory. However, this logical approach to complexity theory is important for a number of reasons, including the following. First, finite model theory provides tools for proving logical inexpressibility results, and a logical inexpressibility result can often be translated into a complexity-theoretic lower bound result. For example, if we could show that the complement of, say, the 3-Colourability Problem (which consists of all those undirected graphs whose vertices can be coloured red, white or blue such that no vertex is joined to another vertex of the same colour), an **NP**-complete problem, could not be defined by a sentence of Σ_1^1 then, by Fagin's Theorem, **NP** would be different from its complementary class **co-NP**; and consequently **NP** would be different from **P**, the class of problems solvable in deterministic polynomial-time (whether **NP** is the same as **co-NP** or **P** are widely regarded as two of the most important and difficult open problems in computer science). Moreover, these tools from finite model theory are not usually available in the complexity-theoretic setting. Second, a logical characterization of a complexity class usually yields new parameters, such as the number of quantifiers or the number of variables in a defining formula. One can restrict these parameters and hope to gain some insight into the actual characterization. Again, the parameters arising are usually not available in the complexity-theoretic setting. Examples of tools from finite model theory are the numerous variants of the well-known Ehrenfeucht-Fraïssé game, first shown by Barwise [5] and Immerman [27] to characterize definability in bounded variable infinitary logic (and developed using earlier results of Ehrenfeucht [14] and Fraïssé [17]); and an example of a new logical parameter is the arity of the quantified second-order relation symbols in a sentence of Σ_1^1. As a matter of fact, combining a variant of the usual Ehrenfeucht-Fraïssé game with a consideration of the class of problems definable by the sentences of existential second-order logic in which the quantified relation symbols are necessarily unary, that is, the class of problems known as monadic **NP**, enabled Fagin to prove that monadic **NP** is not closed under complementation [16]. Thus, whilst whether **NP** = **co-NP** remains unresolved, we do know that monadic **NP** $\neq$ monadic **co-NP**.

The complexity class **NP** can be regarded as a boundary point in the sense that all logical characterizations of complexity classes contained in (but for which the

expectation is that they are different from) **NP** are not as satisfactory as that yielded for **NP** from Fagin's Theorem; for these characterizations only hold in the presence of some built-in relation or relations (equivalently, on a specific class of finite structures) such as a successor relation. For example, the characterization of **P** as the class of problems definable by the sentences of least fixed point logic, due to Immerman [28] and Vardi [49], no longer holds in the absence of a built-in successor relation. The same can be said for many other logical characterizations of **P** such as alternating transitive closure logic [29] or path system logic [46]. Similarly, Immerman's characterization of the complexity class **NL**, the class of problems solvable in non-deterministic logspace, as the class of problems definable by the sentences of transitive closure logic [29] no longer holds in the absence of a built-in successor relation.

Of course, whilst we may lose a complexity-theoretic characterization in the absence of built-in relations, this does provide additional motivation for the consideration of the 'pure' logic (with any built-in relations removed) in the hope that the loss of the complexity-theoretic link might make the logic more amenable to non-expressibility results. One example of such a circumstance is Grädel and McColm's result [22] that there are problems definable in transitive closure logic which can not be defined in deterministic transitive closure logic. In the presence of a built-in successor relation, these two logics capture **NL** and **L** (the class of problems solvable in deterministic logspace), respectively [29], and it is a longstanding open problem in complexity theory as to whether **L** is equal to **NL**. However, the loss of a complexity-theoretic link does not always make life easier: witness Abiteboul and Vianu's result [2] that least fixed point logic has the same expressibility as partial fixed point logic (in the absence of any built-in relations) if, and only if, **P** equals **PSPACE** (in the presence of a built-in successor relation, least fixed point logic captures **P** [28, 49] and partial fixed point logic captures **PSPACE** [1]).

It is the inexpressibility results within transitive closure logic (without a built-in successor relation) due to Grädel [21] and Grädel and McColm [22] that provide some motivation for the research presented here. In [21], it is shown that an infinite (proper) hierarchy of logics, obtained by interleaving applications of the TC operator and applications of the universal quantifier, exists within (the positive fragment of) transitive closure logic. In [22], a powerful result is proven linking (in)expressibility in certain fragments of transitive closure logic and bounded variable infinitary logic, one corollary of which is that there is an infinite (proper) hierarchy within transitive closure logic obtained by interleaving applications of the transitive closure operator and negation. Grädel and McColm also solve a problem first posed by Immerman [29] and show that there are problems in transitive closure logic which are not definable in the positive fragment of transitive closure logic. All these inexpressibility results are proven by playing Ehrenfeucht-Fraïssé games specifically designed for transitive closure logic and bounded variable infinitary logic (the whole issue of the existence of Ehrenfeucht-Fraïssé games to capture definability in a variety of logics has been considered in [34]). This is not unusual as almost all inexpressibility results in finite model theory have been obtained by playing games of one sort or another, usually variants of Ehrenfeucht-Fraïssé games.

Having provided some (albeit spartan) background as to the results from finite model theory which motivate us and the tools which have been used to establish them, let us change tack slightly. We have seen how logics have been developed so as

to capture complexity classes: let us now adopt a somewhat different approach and instead of developing logic to tie in with complexity theory, let us work with a model of computation that is amenable to logical analysis yet is closer to the general notion of a program than a logical formula is. That is, we work with program schemes. Program schemes were extensively studied in the seventies (for example, see [6, 8, 18, 40]), without much regard being paid to an analysis of resources, before a closer complexity analysis was undertaken in, mainly, the eighties (for example, see [24, 32, 47]). There are connections between program schemes and logics of programs, especially dynamic logic [11, 35]. Program schemes have since been further developed to work on finite structures [43], mindful of advances in finite model theory, and it is with a generalization of a specific class of these program schemes that we begin our studies with here.

We begin by defining an infinite hierarchy of program schemes, NPS, whose first level is a class of non-deterministic while-programs where the tests within while instructions are quantifier-free first-order formulae (such program schemes originate in [43]). The next level consists of the closure of these program schemes under universal quantification; the subsequent level consists of non-deterministic while-programs where the tests within while instructions are program schemes from the preceding level; and so on. We show that, in fact, the class of problems accepted by program schemes from the hierarchy is nothing more than the class of problems definable by the sentences of transitive closure logic; and so our seemingly disparate threads, those of definability in logics such as transitive closure logic and solvability by program schemes, begin to tie together. Whilst this first result is nothing startling (and indeed can easily be established), we then go on to show that our hierarchy of program schemes is proper by, essentially, considering program scheme computations on appropriately constructed structures. So it is that we re-create hierarchy results in transitive closure logic similar to those of Grädel [21].

In fact, a result of Grädel and McColm in [22] can be used to obtain our hierarchy results; but only up to a point. Their result only yields hierarchies over a fixed signature when this signature contains 3 binary relation symbols and 2 constant symbols. We prove that these hierarchies remain proper even when we only consider problems involving undirected trees or problems involving out-trees. As well as obtaining refined and more precise hierarchy results, in comparison with those obtained by applying Grädel and McColm's result, what is important in our exposition is that our results are all established *not* using (variants of the usual) Ehrenfeucht-Fraïssé games but by simply considering appropriate computations in program schemes. We believe our presentation to be much more straight-forward, concrete and clear than those in [21] and [22] (although, to be fair, there is more to these two papers than we have mentioned here). For, as exponents of the art well know, it is often difficult to develop winning strategies in even the basic Ehrenfeucht-Fraïssé game, never mind in (generalized) Ehrenfeucht-Fraïssé games adapted to, for example, transitive closure logic. Consequently, our proofs have a significant pedagogic advantage.

However, where we gain some real advantage is in our adoption of a high-level programming formalism as our model, as such a stance enables us to extend our program schemes with a high-level programming construct, the stack (a simple extension using arrays was considered in [43] but only in the presence of a built-in successor relation). Such an extension would not have been available had we remained within

transitive closure logic (not without 'behaving unnaturally' which we would never have been tempted to do). We show that extending the program schemes of NPS with a stack in a natural fashion, to obtain the class of program schemes NPSS, is a real extension in the sense that there are (**P**-complete) problems accepted by program schemes of NPSS which are not accepted by any program scheme of NPS. Moreover, we show that the class of problems accepted by the program schemes of NPSS has an equivalent formulation as path system logic (first studied in [46], in the presence of a built-in successor relation). We go on to show that there are proper infinite hierarchies within NPSS and path system logic mirroring the infinite hierarchies within NPS and transitive closure logic established earlier; and which, again, remain proper even when we only consider problems involving undirected trees or problems involving out-trees. The same comments can also be made about our hierarchies in NPSS and path system logic as were made about the hierarchies in NPS and transitive closure logic, with regard to the applicability of Grädel and McColm's result from [22] (see above). However, again crucially, we establish our results by considering the computations of program schemes on appropriate structures, and without any mention of Ehrenfeucht-Fraïssé games.

We have compared our computational approach with Ehrenfeucht-Fraïssé games above, as (variants of) such games are the most commonly used means for establishing inexpressibility results in finite model theory (of course, other kinds of games have also been played in model theory: see, for example, [25]). However, an approach not dissimilar to our own has previously been undertaken. In [38], McColm develops games for least fixed point logic by considering a sentence (of least fixed point logic) as a program (or, as he puts it, a *rulebook*) so that winning strategies in the game correspond to particular structures satisfying the sentence (and vice versa, in an Ehrenfeucht-Fraïssé style). McColm uses these games to exhibit a proper infinite hierarchy within least fixed point logic, obtained by bounding the number of quantifier alternations. McColm's methodology sits somewhere between our 'purely computational' approach and the usual Ehrenfeucht-Fraïssé style approach in that programs (similar to our program schemes) appear within his methodology but his techniques are still game-theoretic in nature and involve a characterization theorem relating winning strategies in games and satisfiability of sentences. It would be interesting to examine a more precise combination of McColm's methodology and our own.

We have one important further remark to make. The lack of a 'bona fide' logic capturing any complexity class contained within **NP**, and especially **P**, has sparked much research (see [39]). Here, by 'bona fide' we mean that the logic should have a recursive syntax (again, see [39]). Our motivation for considering the class of problems accepted by the program schemes of NPS and NPSS is not to try and derive some 'logical' characterization of **P** or to extend the class of problems within **P** captured by a bona fide logic. We are interested in the classes of program schemes NPS and NPSS as resource-bounded models of computation in their own right. Of course, our interest has been further stimulated given the results in this paper establishing a relationship between these classes of program schemes and logics previously studied in finite model theory.

This paper is organised as follows. In the next section, Section 2, we give definitions of the fundamental concepts and logics from finite model theory pertinent

to this paper (a general reference is [13] within which the reader will find explicit definitions of concepts which, although mentioned here, are not absolutely essential to our account). In Section 3, we introduce our class of program schemes NPS and tie together NPS and transitive closure logic. In Section 4, we detail the general construction used to build the structures from which we obtain our inexpressibility results, with these basic inexpressibilty results proven in Section 5. Also in Section 5, we apply our inexpressibility results to the program schemes of NPS and other logics to yield a number of hierarchy results. In Section 6, we explain how a stack can be added to our program schemes to yield the class of program schemes NPSS, and we characterize NPSS as path system logic. In Section 7, after highlighting Grädel and McColm's main result from [22] and how it can be applied (see above), we obtain proper hierarchies in NPSS and path system logic. Finally, in Section 8, we present our conclusions and some directions for further research.

2 Preliminaries

Throughout, a *signature* σ is a tuple $\langle R_1, \ldots, R_r, C_1, \ldots, C_c \rangle$, where each R_i is a relation symbol, of arity a_i, and each C_j is a constant symbol: in the case that σ consists only of relation symbols, we say that σ is *relational*. If σ and σ' are two signatures having no symbol with the same name then $\sigma \cup \sigma'$ consists of the signature whose symbols are those of σ in union with those of σ'. *First-order logic over some signature* σ, $\mathrm{FO}(\sigma)$, consists of those formulae built from atomic formulae over σ using $\wedge$, $\vee$, $\neg$, $\forall$ and $\exists$; and $\mathrm{FO} = \cup\{\mathrm{FO}(\sigma) : \sigma \text{ is some signature}\}$.

A *finite structure* $\mathcal{A}$ *over the signature* σ, or σ*-structure*, consists of a finite *universe* or *domain* $|\mathcal{A}|$ together with a relation R_i of arity a_i for every relation symbol R_i of σ, and a constant $C_j \in |\mathcal{A}|$ for every constant symbol C_j (by an abuse of notation, we do not distinguish between constants or relations and constant or relation symbols). A finite structure $\mathcal{A}$ whose domain consists of n distinct elements has *size* n, and we denote the size of $\mathcal{A}$ by $|\mathcal{A}|$ also (this does not cause confusion). We only ever consider structures of size at least 2, and the class of all finite structures over the signature σ of size at least 2 is denoted $\mathrm{STRUCT}(\sigma)$. A *problem* over some signature σ consists of a subset of $\mathrm{STRUCT}(\sigma)$ which is closed under isomorphism; that is, if $\mathcal{A}$ is in the problem then so is every isomorphic copy of $\mathcal{A}$. Throughout, all our structures will be finite. If $\mathcal{A}$ and $\mathcal{B}$ are two structures over the same signature σ such that $|\mathcal{A}| \subseteq |\mathcal{B}|$ and such that the restriction of $\mathcal{B}$ to $|\mathcal{A}|$ is isomorphic to $\mathcal{A}$ (and so, for one thing, every constant of $\mathcal{B}$ is in $|\mathcal{A}|$) then we write $\mathcal{A} \subseteq \mathcal{B}$.

We are now in a position to consider the class of problems defined by the sentences of FO: we denote this class of problems by FO also, and do likewise for other logics. It is widely acknowledged that as a means for defining problems, first-order logic leaves a lot to be desired especially when we have in mind developing a relationship between computational complexity and logical definability (see, for example, [13]). Consequently, we now give one way of increasing the expressibility of FO: augmenting FO with (uniform or vectorized sequences of) Lindström quantifiers. There are, of course, other ways to increase the expressibility of FO: we have already mentioned second-order logic, least fixed point logic and bounded variable infinitary logic. Whilst we shall (briefly) meet bounded variable infinitary logic later, we concentrate here on how we extend FO using Lindström quantifiers (and refer the reader to [13] for details

regarding the other logics).

Define the signature $\sigma_{2++} = \langle E, C, D \rangle$, where E is a binary relation symbol and C and D are constant symbols, and define the problem TC as

$$\mathrm{TC} = \{\mathcal{A} \in \mathrm{STRUCT}(\sigma_{2++}) : \text{ the digraph with vertex set } |\mathcal{A}| \text{ and edge set given by the relation } E \text{ contains a path from vertex } C \text{ to vertex } D\}.$$

Corresponding to the problem TC is an operator of the same name; more precisely, an infinite uniform, or vectorized, sequence of *Lindström quantifiers* (whilst we do not define here explicitly what a Lindström quantifier is, we hope that the essence of Lindström quantifiers is gleaned from what follows: again, see [13]). The logic $(\pm\mathrm{TC})^*[\mathrm{FO}]$, or *transitive closure logic*, is the closure of FO under the usual first-order connectives and quantifiers, and also the operator TC, with TC applied as follows.

Given a formula $\varphi(\mathbf{x}, \mathbf{y}) \in (\pm\mathrm{TC})^*[\mathrm{FO}]$, where the variables of the k-tuples $\mathbf{x}$ and $\mathbf{y}$, for some k, are all distinct and free in φ, the formula Φ defined as $\mathrm{TC}[\lambda\mathbf{x}, \mathbf{y}\varphi](\mathbf{u}, \mathbf{v})$, where $\mathbf{u}$ and $\mathbf{v}$ are k-tuples of (not necessarily distinct) constant symbols and variables, is also a formula of $(\pm\mathrm{TC})^*[\mathrm{FO}]$, with the free variables of Φ being those variables in $\mathbf{u}$ and $\mathbf{v}$, as well as the free variables of φ different from those in the tuples $\mathbf{x}$ and $\mathbf{y}$. If Φ is a sentence then it is interpreted in a structure $\mathcal{A} \in \mathrm{STRUCT}(\sigma)$, where σ is the underlying signature, as follows. We build a digraph with vertex set $|\mathcal{A}|^k$ and edge set

$$\{(\mathbf{a}, \mathbf{b}) \in |\mathcal{A}|^k \times |\mathcal{A}|^k : \varphi(\mathbf{a}, \mathbf{b}) \text{ holds in } \mathcal{A}\},$$

and say that $\mathcal{A} \models \Phi$ if, and only if, there is a path in this digraph from vertex $\mathbf{u}$ to vertex $\mathbf{v}$ (the semantics can easily be extended to arbitrary formulae of $(\pm\mathrm{TC})^*[\mathrm{FO}]$: see, for example, [13] for a more detailed semantic definition).

We occasionally focus on some fragments of $(\pm\mathrm{TC})^*[\mathrm{FO}]$.

- $\mathrm{TC}^*[\mathrm{FO}]$ consists of all those formulae where applications of TC do not appear within the scope of a negation sign.
- $(\pm\mathrm{TC})^i[\mathrm{FO}]$ consists of all those formulae where at most i applications of TC may be nested.
- $\mathrm{TC}^i[\mathrm{FO}]$ consists of all those formulae where applications of TC do not appear within the scope of a negation sign and where at most i applications of TC may be nested.

We reiterate that TC is essentially an infinite sequence of Lindström quantifiers $\{\mathrm{TC}_k\}$ where TC_k is the corresponding quantifier which binds $2k$ free variables in the formula to which it is applied. In a celebrated result, Immerman [29, 30] captured the complexity class **NL** by the logic $(\pm\mathrm{TC})^*[\mathrm{FO}]$, but only in the presence of a built-in successor relation (more later), thus obtaining as a corollary that **NL** = **co-NL**.

One can augment FO with an operator (or operators) such as TC corresponding to *any* problem (or problems) and examine the class of problems so captured. A variety of such logics have been formed and many well-known complexity classes

subsequently captured (see, for example, the presentation and references in [46]). Of particular interest to us in this paper is the logic formed by extending FO using an operator corresponding to the problem PS defined below.

A *path system* is a set of vertices and a set of *rules* of the form $(x, y) \mapsto z$, where x, y and z are vertices, together with a distinguished *source* vertex and a distinguished *sink* vertex. The set of *accessible vertices* is built by initially assuming the source vertex to be accessible and then continually applying the rules until the current set of accessible vertices can be made no bigger, via: 'if the vertices x and y have been shown to be accessible and $(x, y) \mapsto z$ then z is accessible' (x and y need not be distinct). Let $\sigma_{3++} = \langle R, C, D \rangle$, where R is a relation symbol of arity 3 and C and D are constant symbols. Any σ_{3++}-structure can clearly be considered as a path system with C the source and D the sink and where the rules are given by $\{(x, y) \mapsto z : R(x, y, z) \text{ holds}\}$. Define

$$\begin{aligned} \text{PS} = \{\mathcal{A} \in \text{STRUCT}(\sigma_{3++}) : &\text{ the path system with vertex set given by} \\ &|\mathcal{A}| \text{ and accessibility rules given by the relation } R \text{ is such that} \\ &\text{the vertex } D \text{ is accessible from the vertex } C\}. \end{aligned}$$

The problem PS has long been known to be complete for **P** via logspace reductions [10], and in [46] the logic $(\pm\text{PS})^*[\text{FO}]$, in the presence of a built-in successor relation, was shown to capture **P**.

3 Program Schemes

In this section we introduce our notion of a program scheme.

Definition 1 A *program scheme* $\rho \in \text{NPS}(1)$ involves a finite set $\{x_1, x_2, \ldots, x_k\}$ of *variables*, for some $k \geq 1$, and is over a signature σ. It consists of a finite sequence of *instructions* where each instruction, apart from the first and the last, is one of the following:

- an *assignment instruction* of the form '$x_i := y$', where $i \in \{1, 2, \ldots, k\}$ and where y is either a variable from $\{x_1, x_2, \ldots, x_k\}$, a constant symbol of σ or one of the special constant symbols 0 and *max* which do not appear in any signature;
- a *guess instruction* of the form 'GUESS x_i', where $i \in \{1, 2, \ldots, k\}$; or
- a *while instruction* of the form 'WHILE t DO $\alpha_1; \alpha_2; \ldots; \alpha_q$ OD', where t is a quantifier-free formula of $\text{FO}(\sigma \cup \{0, max\})$ whose free variables are from $\{x_1, x_2, \ldots, x_k\}$ and where each of $\alpha_1, \alpha_2, \ldots, \alpha_q$ is another instruction of one of the three forms given here (note that there may be nested while instructions).

The first instruction of ρ is 'INPUT$(x_1, x_2, \ldots, x_l)$' and the last instruction is 'OUTPUT$(x_1, x_2, \ldots, x_l)$', for some l where $1 \leq l \leq k$. The variables $x_1, x_2, \ldots, x_l$ are the *input-output variables* of ρ, the variables $x_{l+1}, x_{l+2}, \ldots, x_k$ are the *free variables* of ρ and, further, any free variable of ρ never appears on the left-hand side

of an assignment instruction nor in a guess instruction. Essentially, free variables appear in ρ as if they were constant symbols. (As we soon see, other types of program scheme might have bound variables: however, no program scheme of NPS(1) has bound variables.)

A program scheme $\rho \in \mathrm{NPS}(1)$ over σ with s free variables, say, takes a σ-structure $\mathcal{A}$ and s additional values from $|\mathcal{A}|$, one for each free variable of ρ, as input; that is, an expansion $\mathcal{A}'$ of $\mathcal{A}$ by adjoining s additional constants. The program scheme ρ computes on $\mathcal{A}'$ in the obvious way except that:

- execution of the instruction 'GUESS x_i' non-deterministically assigns an element of $|\mathcal{A}|$ to the variable x_i;
- the constants 0 and max are interpreted as two arbitrary but distinct elements of $|\mathcal{A}|$; and
- initially, every input-output variable is assumed to have the value 0.

Note that throughout a computation of ρ, the value of any free variable does not change. The expansion $\mathcal{A}'$ of the structure $\mathcal{A}$ is *accepted* by ρ, and we write $\mathcal{A}' \models \rho$, if, and only if, there exists a computation of ρ on this expansion such that the output-instruction is reached with all input-output variables having the value max.

We want the sets of structures accepted by our program schemes to be problems, i.e., closed under isomorphisms, and so we only ever consider program schemes ρ (including those defined in Definition 1 and in future) where a structure is accepted by ρ when 0 and max are given two distinct values from the universe of the structure if, and only if, it is accepted no matter which pair of distinct values is chosen for 0 and max. Let us reiterate: when we say that ρ is a program scheme of, for example, NPS(1) we mean that ρ accepts a problem and the acceptance of any input structure does not depend upon the pair of distinct values we give to 0 and max.

Compare the above stipulation with the usual situation in logic. It is generally accepted that the syntax of any logic should be recursive; that is, the set of well-formed formulae should be recursive (see [39]). Analogously, we might expect that a class of program schemes should be recursively enumerable. Trakhtenbrot's Theorem ([48]: see also [13, Theorem 6.2.1]) tells us that it is undecidable as to whether an arbitrary first-order sentence holds in every (appropriate) finite structure. It is conceivable that (but, as far as we know, unknown) whether a program scheme satisfies our criterion (above, regarding 0 and max) is undecidable too. Hence, if we follow the accepted practice in logic then we may have some difficulties with whether our class of program schemes is 'bona fide' or not. However, we could easily circumvent this (possibly non-existent) difficulty by, for example, insisting that our input-output variables are of a different, *Boolean* type, only taking the values 'true' or 'false', and use these variables to signal acceptance or rejection (we do not go into details). Consequently, we leave the definition of our program schemes as it stands (safe in the knowledge that we could force it to conform to standard practice if required). We return to 0 and max later when they appear in logics in an analogous fashion.

Remark 2 (a) We can easily build the usual 'if' and 'if-then-else' instructions using while instructions (see, for example, [43]).

(b) Our program schemes (including those defined above and in future) may be such that certain computations do not terminate.

Example 3 Let the program scheme $\rho \in \mathrm{NPS}(1)$ over σ_{2++} be defined as follows.

1. INPUT(x_1, x_2)
2. x_1 := C
3. WHILE $x_1 \neq D$ DO
4. GUESS x_2
5. WHILE $\neg E(x_1, x_2)$ DO
6. GUESS x_2 OD
7. x_1 := x_2 OD
8. x_1 := max
9. x_2 := max
10. OUTPUT(x_1, x_2)

(We present program schemes in an indented style to aid readability.) Then ρ does indeed accept a problem, and the problem accepted by ρ is TC.

Example 4 The signature $\sigma_{2,2} = \langle P, N \rangle$, where P and N are binary relation symbols. We think of a $\sigma_{2,2}$-structure of size n as a conjunction of clauses of Boolean literals as follows. For convenience, rename the elements of the domain as $0, 1, \ldots, n-1$. There are n clauses $C_0, C_1, \ldots, C_{n-1}$ (some of which might be empty) and there are n Boolean variables $X_0, X_1, \ldots, X_{n-1}$. The literal X_i is in clause C_j if, and only if, $P(i,j)$ holds, and the literal $\neg X_i$ is in clause C_j if, and only if, $N(i,j)$ holds. Empty clauses are satisfiable by definition. The problem SAT is defined as

$$\{\mathcal{A} \in \mathrm{STRUCT}(\sigma_{2,2}) : \text{ the set of clauses } \mathcal{A} \text{ is satisfiable}\}.$$

In [31], the following result was proven.

Proposition 5 Let $\mathcal{C}$ be some set of clauses, over the set of Boolean variables B, containing 0 or 2 distinct literals. Let G be the digraph whose vertex set is the set of literals over B and whose edge set is

$$\{(l, \neg l') : \text{ there is a clause } \{l \vee l'\} \text{ in } \mathcal{C} \text{ where } l \text{ and } l' \text{ are literals}\}$$

($\neg\neg l$ is identified with l). Then the set of clauses $\mathcal{C}$ is not satisfiable if, and only if, there is a path in the digraph G from l to $\neg l$ and also one from $\neg l$ to l, for some literal l.

Consider the following program scheme $\rho \in \mathrm{NPS}(1)$ over $\sigma_{2,2}$ when we only allow inputs $\mathcal{A}$ where $\mathcal{A}$ is such that every clause has 0 or 2 distinct literals. Then ρ accepts those structures of this type that are not satisfiable.

1. INPUT$(x_1, x_2, x_3, x_4, x_5, x_6)$
2. GUESS x_1
3. $(x_2, x_3) := (0, x_1)$
4. WHILE $\neg(x_2 = max \wedge x_3 = x_1)$ DO
5. GUESS x_4, x_5

6. GUESS x_6
7. IF $(x_2 \neq x_4 \vee x_3 \neq x_5) \wedge ((x_2 = 0 \wedge P(x_3, x_6)) \vee (x_2 = max \wedge N(x_3, x_6))$ $\wedge((x_4 = 0 \wedge P(x_5, x_6)) \vee (x_4 = max \wedge N(x_5, x_6))$ THEN
8. $x_3 := x_5$
9. IF $x_4 = 0$ THEN
10. $x_2 := max$ ELSE
11. $x_2 := 0$ FI OD
12. $(x_2, x_3) := (max, x_1)$
13. WHILE $\neg(x_2 = 0 \wedge x_3 = x_1)$ DO
14. GUESS x_4, x_5
15. GUESS x_6
16. IF $(x_2 \neq x_4 \vee x_3 \neq x_5) \wedge ((x_2 = 0 \wedge P(x_3, x_6)) \vee (x_2 = max \wedge N(x_3, x_6))$ $\wedge((x_4 = 0 \wedge P(x_5, x_6)) \vee (x_4 = max \wedge N(x_5, x_6))$ THEN
17. $x_3 := x_5$
18. IF $x_4 = 0$ THEN
19. $x_2 := max$ ELSE
20. $x_2 := 0$ FI OD
21. $(x_1, x_2, x_3, x_4, x_5, x_6) := (max, max, max, max, max, max)$
22. OUTPUT$(x_1, x_2, x_3, x_4, x_5, x_6)$

(The shorthand used above should be obvious.) Essentially, we guess a Boolean literal X_{x_1}, the first while loop checks to see whether there is a path in G from X_{x_1} to $\neg X_{x_1}$, and the second while loop checks to see whether there is a path in G from $\neg X_{x_1}$ to X_{x_1} (where G is the digraph as in Proposition 5). The current vertex in G, a literal, is encoded as (x_2, x_3) where if $x_2 = 0$ then the literal is X_{x_3} and if $x_2 = max$ then the literal is $\neg X_{x_3}$.

The class of program schemes NPS(1) can be regarded as a very basic class of nondeterministic program schemes based on while loops. An important point to note is that whereas the usual existential quantifier is catered for via the guess instruction (intuitively speaking), there is no such analogous modelling of the universal quantifier. Consequently, we extend these basic program schemes by introducing universal quantification in the following manner.

Definition 6 Let σ be some signature. For some $m \geq 1$, let the program scheme $\rho \in \text{NPS}(2m-1)$ be over the signature σ and involve the variables $x_1, x_2, \ldots, x_k$. Suppose that the variables $x_1, x_2, \ldots, x_l$ are the input-output variables of ρ, that the variables $x_{l+1}, x_{l+2}, \ldots, x_{l+s}$ are the free variables and that the remaining variables are the bound variables. Let $x_{i_1}, x_{i_2}, \ldots, x_{i_p}$ be free variables of ρ, for some p such that $1 \leq p \leq s$. Then

$$\forall x_{i_1} \forall x_{i_2} \ldots \forall x_{i_p} \rho$$

is a program scheme of NPS($2m$), which we denote by ρ', with no input-output variables, with free variables those of $\{x_{l+1}, x_{l+2}, \ldots, x_{l+s}\} \setminus \{x_{i_1}, x_{i_2}, \ldots, x_{i_p}\}$ and with the remaining variables of $\{x_1, x_2, \ldots, x_k\}$ as its bound variables.

A program scheme such as ρ' takes expansions $\mathcal{A}'$ of σ-structures $\mathcal{A}$ by adjoining $s-p$ constants as input (one for each free variable), and ρ' accepts such an expansion $\mathcal{A}'$ if, and only if, for every expansion $\mathcal{A}''$ of $\mathcal{A}'$ by p additional constants (one for each variable x_{i_j}), $\mathcal{A}'' \models \rho$.

Definition 7 Let σ be some signature. A program scheme $\rho' \in \mathrm{NPS}(2m-1)$, for some $m \geq 2$, over the signature σ and involving the variables of $\{x_1, x_2, \ldots, x_k\}$, is formed exactly as are the program schemes of NPS(1), with the input-output and free variables defined accordingly, except that the test in some while instruction is a program scheme $\rho \in \mathrm{NPS}(2m-2)$ whose free and bound variables are all from $\{x_1, x_2, \ldots, x_k\}$ (note that ρ has no input-output variables). However, there are further stipulations:

- all free variables in any test $\rho \in \mathrm{NPS}(2m-2)$ in any while instruction are input-output or free variables of ρ';
- the bound variables of ρ' consist of all bound variables of any test $\rho \in \mathrm{NPS}(2m-2)$ in any while instruction (and no bound variable is ever an input-output or free variable of ρ); and
- this accounts for all variables of $\{x_1, x_2, \ldots, x_k\}$.

Of course, any free variable of ρ' never appears on the left-hand side of an assignment instruction or in a guess instruction.

A program scheme $\rho' \in \mathrm{NPS}(2m-1)$ takes expansions $\mathcal{A}'$ of σ-structures $\mathcal{A}$ by adjoining s constants as input, where s is the number of free variables, and computes on $\mathcal{A}'$ in the obvious way except that when some while instruction is encountered, the test, which is a program scheme $\rho \in \mathrm{NPS}(2m-2)$, is evaluated according to the expansion of $\mathcal{A}'$ by the current values of any relevant input-output variables of ρ' (which may be free in ρ).

Remark 8 A simple analysis yields that we can build the usual 'if' and 'if-then-else' instructions in the program schemes of NPS(m), for all odd $m \geq 1$. Indeed, henceforth we assume that these instructions are at our disposal.

Example 9 Let $\sigma = \langle E, U \rangle$, where E is a binary relation symbol and U is a unary relation symbol. A σ-structure can be envisaged as a digraph, whose edge relation is given by E, with a specified set of vertices, given by U, called *roots*. The following program scheme $\rho' \in \mathrm{NPS}(3)$ accepts the problem consisting of those rooted digraphs for which at least one of the roots is such that there are paths from the root to every other vertex.

1. INPUT(x_1)
2. GUESS x_1
3. WHILE $\neg U(x_1)$ DO
4. GUESS x_1 OD
5. IF $\forall x_2 \rho(x_1, x_2)$ THEN
6. $x_1 := max$ ELSE
7. $x_1 := 0$ FI
8. OUTPUT(x_1)

where the program scheme $\rho \in \mathrm{NPS}(1)$ is defined as

1. INPUT(x_3, x_4)
2. $x_3 := x_1$

3. WHILE $x_3 \neq x_2$ DO
4. GUESS x_4
5. IF $E(x_3, x_4)$ THEN
6. $x_3 := x_4$ FI OD
7. $(x_3, x_4) := (max, max)$
8. OUTPUT(x_3, x_4)

(The input-output variables of ρ are x_3 and x_4, the free variables are x_1 and x_2 and there are no bound variables. The input-output variable of ρ' is x_1, there are no free variables and the bound variables are x_2, x_3 and x_4.)

In order to facilitate our understanding of program schemes and their computations, we shall henceforth abuse Definition 1 as follows. Whereas, in Definition 1, we talked of an instruction of the form 'WHILE t DO $\alpha_1; \alpha_2; \ldots; \alpha_q$ OD', throughout the rest of this paper we shall think of such an instruction as a sequence of instructions, the first of which is an evaluation of t, the second of which is α_1 (or possible a sequence of instructions corresponding to α_1 if α_1 is of the form 'WHILE ... DO ... OD'), the third of which is α_2, and so on. That is, we shall think of every program scheme as being written in the (programming language) style of the preceding examples, with computations defined accordingly (another abuse is that we sometimes group assignments together to form one instruction, as in instruction 7 of the program scheme ρ in Example 9). Thus, in future when we say 'instruction' we mean assignments, guesses and the evaluation of tests (where the test itself might possibly involve another program scheme), and we label these instructions as we have done in previous examples. In consequence, we envisage computations of program schemes of NPS(1) as being sequences of tuples consisting of: (a) values of the input-output variables; and (b) an integer denoting the label of the next instruction to be executed. We shall expand upon this point later.

As the reader might have guessed, there is a close relationship between our classes of program schemes and transitive closure logic.

Definition 10 Define:

- $\pm$TC(1) to be the set of formulae of the form

$$\mathrm{TC}[\lambda \mathbf{x}, \mathbf{y} \psi](\mathbf{u}, \mathbf{v}),$$

where ψ is quantifier-free first-order and where $\mathbf{u}$ and $\mathbf{v}$ are tuples of constant symbols or variables;

- $\pm$TC$(m+1)$, for odd $m \geq 1$, to be the universal closure of TC(m), i.e., the set of formulae of the form $\forall z_1 \ldots \forall z_k \psi$, where ψ is a formula of TC(m); and

- $\pm$TC$(m+1)$, for even $m \geq 2$, to be the set of formulae of the form

$$\mathrm{TC}[\lambda \mathbf{x}, \mathbf{y}(\psi_1 \vee \neg \psi_2)](\mathbf{u}, \mathbf{v}),$$

where $\psi_1, \psi_2 \in \pm\mathrm{TC}(m)$ and where $\mathbf{u}$ and $\mathbf{v}$ are tuples of constant symbols or variables (not necessarily distinct).

Note that in order to form (non-trivial) sentences in $\pm\mathrm{TC}(m)$, for m odd, and so consider $\pm\mathrm{TC}(m)$ as a class of (non-trivial) problems, we need at least two distinct constant symbols. Consequently, when we talk about $\pm\mathrm{TC}(m)$, we assume that there are always two (distinct) *built-in constants* available, 0 and max, and we only consider sentences φ of $\pm\mathrm{TC}(m)$ for which the following is true: for any (appropriate) structure $\mathcal{A}$, $\mathcal{A} \models \varphi$ with 0 and max given distinct values if, and only if, $\mathcal{A} \models \varphi$ no matter which pair of distinct values are taken for 0 and max. That is, we proceed as we did for 0 and max in our program schemes.

An alternative would be to only consider signatures containing the constant symbols 0 and max and structures for which 0 and max are interpreted differently, as is done in [13, 21, 22], for example. However, we prefer to work with built-in constant symbols for two reasons. First, it may be the case that the natural encoding of a problem (involving, for example, graphs) as a set of structures does not involve any constant symbols. We feel that including constant symbols in a signature unnecessarily and treating the corresponding constants in a structure as essential to the problem instance is unsatisfactory (see Garey and Johnson's discussion on reasonable complexity-theoretic encoding schemes in [19]). Second, including 0 and max as built-in constant symbols is in keeping with how one generally introduces an ordering into structures in descriptive complexity theory: see, for example, [13, Section 6.5] where a built-in successor relation is introduced into a logic in the same way that our built-in constants have been introduced (the phraseology in [13] is actually that ordered representations of structures are considered but this amounts to the same thing as saying that there is a built-in successor relation available). However, be this as it may, the results in [13, 21, 22] and in this paper hold whether we assume the existence of built-in constants or we only work with signatures containing the constant symbols 0 and max (and where these symbols are always interpreted differently).

A simple induction (similar to those in [43], for example) yields the following result in which we identify, as we do throughout, a class of program schemes (resp. a logic) with the problems accepted by the program schemes (resp. defined by the sentences of the logic). We write $\mathrm{NPS} = \cup\{\mathrm{NPS}(m) : m \geq 1\}$.

Theorem 11 In the presence of two built-in constants, $\mathrm{NPS}(m) = \pm\mathrm{TC}(m)$, for every $m \geq 1$: consequently,

$$\mathrm{NPS} = (\pm\mathrm{TC})^*[\mathrm{FO}].$$

Note that $\mathrm{NPS} = (\pm\mathrm{TC})^*[\mathrm{FO}]$ even in the absence of two built-in constants in transitive closure logic as we can 'build two distinct constants' by existential quantification.

In the presence of a built-in *successor relation* and two built-in constants, 0 and max, denoting the minimum and the maximum with respect to the successor relation, i.e., a binary relation $\{(a_0, a_1), (a_1, a_2), \ldots, (a_{n-2}, a_{n-1})\}$ in a structure of size n where all the a_i's are distinct and $a_0 = 0$ and $a_{n-1} = max$, it is well-known that everything collapses; and that this collapse is to **NL**.

Theorem 12 [29, 30, 43] In the presence of a built-in successor relation,

$$\mathrm{NPS} = \mathrm{NPS}(1) = (\pm\mathrm{TC})^*[\mathrm{FO}] = \mathrm{TC}^1[\mathrm{FO}] = \mathbf{NL}.$$

4 Constructing suitable structures

We now detail a general construction which yields structures suitable for proving hierarchy results in classes of program schemes and logics. This construction is essentially a generalization of that in [21] (a similar construction is also given in [13]), which in turn is derived from a construction in [7].

Let σ be some relational signature containing the unary relation symbol U_0 and let $\mathcal{A}^0 \subseteq \mathcal{B}^0$ be σ-structures such that

$$|\{u \in |\mathcal{A}^0| : U_0(u) \text{ holds in } \mathcal{A}^0\}| = |\{u \in |\mathcal{B}^0| : U_0(u) \text{ holds in } \mathcal{B}^0\}| = 1.$$

Fix $m \geq 1$ and let $U_1, U_2, \ldots, U_m$ be unary relation symbols not in σ. If there is a binary relation symbol E in σ then define the signature $\sigma^m = \sigma \cup \{U_1, \ldots, U_m\}$; otherwise define the signature $\sigma^m = \sigma \cup \{E, U_1, \ldots, U_m\}$.

For any $k \geq 1$, the σ^1-structure $\mathcal{A}^1_k$ is built from $\mathcal{A}^0$ and $\mathcal{B}^0$ as follows.

- Take $k+1$ copies of $\mathcal{B}^0$ and 1 copy of $\mathcal{A}^0$ (all copies are disjoint) and introduce a new element v; hence, the size of the universe of $\mathcal{A}^1_k$ is $|\mathcal{A}^0| + (k+1)|\mathcal{B}^0| + 1$.
- For any relation symbol R of $\sigma \setminus \{E\}$, the relation R of $\mathcal{A}^1_k$ is the union of the relations R of the copies of $\mathcal{A}^0$ and $\mathcal{B}^0$.
- The relation U_1 of $\mathcal{A}^1_k$ is $\{v\}$.
- The relation E of $\mathcal{A}^1_k$ is the union of the relations E of the copies of $\mathcal{A}^0$ and $\mathcal{B}^0$, in union with $\{(v,u),(u,v) : U_1(v) \text{ and } U_0(u) \text{ hold in } \mathcal{A}^1_k\}$.

The σ^1-structure $\mathcal{B}^1_k$ is built as is $\mathcal{A}^1_k$ except that the copy of $\mathcal{A}^0$ is replaced with another copy of $\mathcal{B}^0$. The structures $\mathcal{A}^1_k$ and $\mathcal{B}^1_k$ can be visualized as in Fig. 1. Note that $\mathcal{A}^1_k \subseteq \mathcal{B}^1_k$ via a natural embedding π (indeed, there are numerous such natural embeddings).

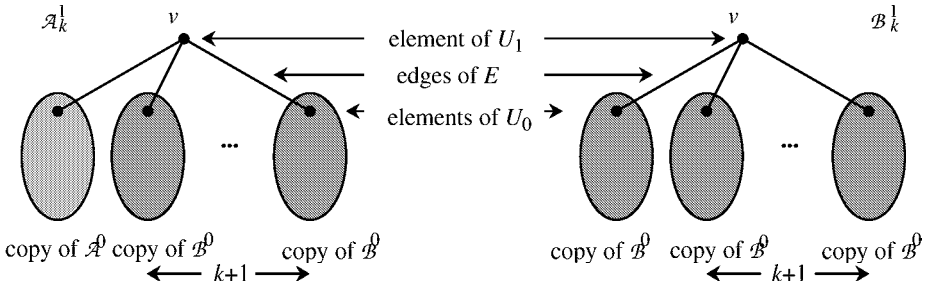

Figure 1. The structures $\mathcal{A}^1_k$ and $\mathcal{B}^1_k$.

For $k \geq 1$ and $m \geq 2$ even, the σ^m-structure $\mathcal{B}^m_k$ is built as follows.

- Take $k+1$ copies of $\mathcal{A}^{m-1}_k$ and 1 copy of $\mathcal{B}^{m-1}_k$ (all copies are disjoint) and introduce a new element v; and so the size of the universe of $\mathcal{B}^m_k$ is $|\mathcal{B}^{m-1}_k| + (k+1)|\mathcal{A}^{m-1}_k| + 1$.
- For any relation symbol R of $\sigma^{m-1} \setminus \{E\}$, the relation R of $\mathcal{B}^m_k$ is the union of the relations R of the copies of $\mathcal{A}^{m-1}_k$ and $\mathcal{B}^{m-1}_k$.

- The relation U_m of $\mathcal{B}_k^m$ is $\{v\}$.
- The relation E of $\mathcal{B}_k^m$ is the union of the relations E of the copies of $\mathcal{A}_k^{m-1}$ and $\mathcal{B}_k^{m-1}$, in union with $\{(v,u),(u,v) : U_m(v)$ and $U_{m-1}(u)$ hold in $\mathcal{B}_k^m\}$.

The σ^m-structure $\mathcal{A}_k^m$ is built as is $\mathcal{B}_k^m$ except that the copy of $\mathcal{B}_k^{m-1}$ is replaced with another copy of $\mathcal{A}_k^{m-1}$. The structures $\mathcal{A}_k^m$ and $\mathcal{B}_k^m$ can be visualized similarly to those in Figs. 2 and 3 (where $\mathcal{A}_k^m$ and $\mathcal{B}_k^m$, for m odd, are depicted). Note that $\mathcal{A}_k^m \subseteq \mathcal{B}_k^m$ via a natural embedding π (again, there are numerous such natural embeddings).

For $k \geq 1$ and $m \geq 3$ odd, the σ^m-structure $\mathcal{A}_k^m$ is built as follows.

- Take $k+1$ copies of $\mathcal{B}_k^{m-1}$ and 1 copy of $\mathcal{A}_k^{m-1}$ (all copies are disjoint) and introduce a new element v; and so the size of the universe of $\mathcal{A}_k^m$ is $|\mathcal{A}_k^{m-1}| + (k+1)|\mathcal{B}_k^{m-1}| + 1$.
- For any relation symbol R of $\sigma^{m-1} \setminus \{E\}$, the relation R of $\mathcal{A}_k^m$ is the union of the relations R of the copies of $\mathcal{A}_k^{m-1}$ and $\mathcal{B}_k^{m-1}$.
- The relation U_m of $\mathcal{A}_k^m$ is $\{v\}$.
- The relation E of $\mathcal{A}_k^m$ is the union of the relations E of the copies of $\mathcal{A}_k^{m-1}$ and $\mathcal{B}_k^{m-1}$, in union with $\{(v,u),(u,v) : U_m(v)$ and $U_{m-1}(u)$ hold in $\mathcal{A}_k^m\}$.

The σ^m-structure $\mathcal{B}_k^m$ is built as is $\mathcal{A}_k^m$ except that the copy of $\mathcal{A}_k^{m-1}$ is replaced with another copy of $\mathcal{B}_k^{m-1}$. The structures $\mathcal{A}_k^m$ and $\mathcal{B}_k^m$ can be visualized as in Figs. 2 and 3. Note that $\mathcal{A}_k^m \subseteq \mathcal{B}_k^m$ via a natural embedding π.

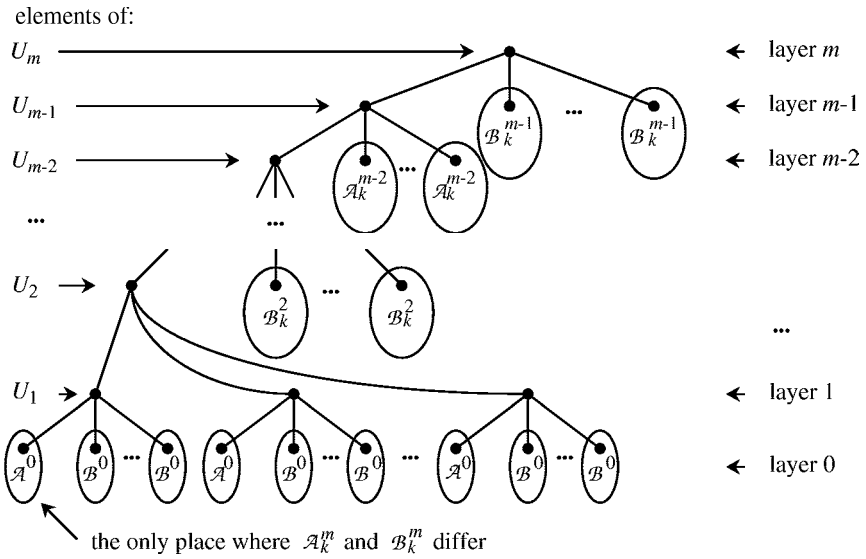

Figure 2. The structure $\mathcal{A}_k^m$ when m is odd.

5 Some hierarchy results

We can now use the structures constructed in the previous section to obtain some hierarchy results in our class of program schemes NPS, and also in some related

logics. For notational convenience, throughout the statement of Theorem 13 and its proof, by $\mathcal{A}_k^0$ and $\mathcal{B}_k^0$ we really mean $\mathcal{A}^0$ and $\mathcal{B}^0$, respectively.

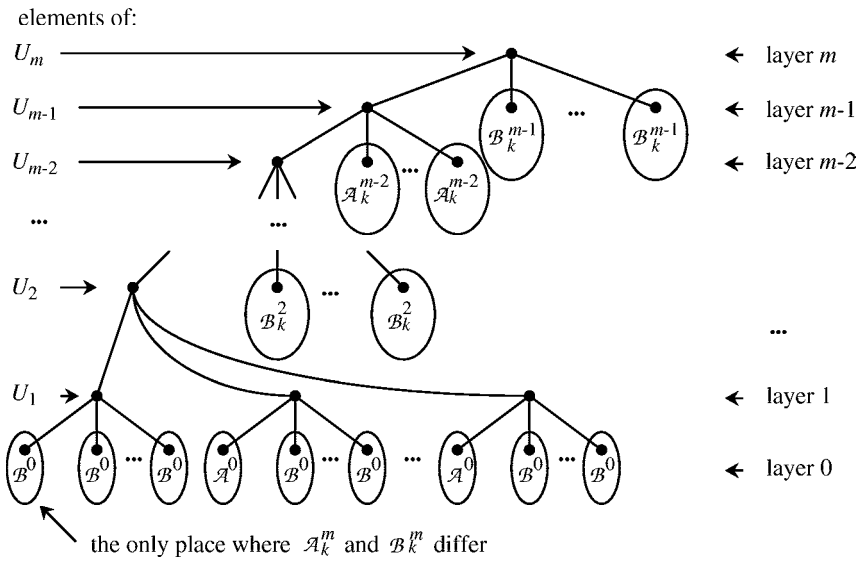

Figure 3. The structure $\mathcal{B}_k^m$ when m is odd.

Theorem 13 Let σ be some relational signature containing the unary relation symbol U_0 and let $\mathcal{A}^0$ and $\mathcal{B}^0$ be σ-structures such that:

- $\mathcal{A}^0 \subseteq \mathcal{B}^0$; and
- $|\{u \in |\mathcal{A}^0| : U_0(u) \text{ holds in } \mathcal{A}^0\}| = |\{u \in |\mathcal{B}^0| : U_0(u) \text{ holds in } \mathcal{B}^0\}| = 1$.

Fix $m \geq 1$, $k \geq 1$ and $r \geq 0$, and:

- let $\rho \in \mathrm{NPS}(m)$ be over the signature σ^{m+r} and involve k variables, s of which are free; and
- let $\tilde{\mathcal{A}}_k^{m+r}$ and $\tilde{\mathcal{B}}_k^{m+r}$ be expansions of the σ^{m+r}-structures $\mathcal{A}_k^{m+r}$ and $\mathcal{B}_k^{m+r}$ by adjoining s constants (one for each free variable of ρ) so that:
  - $\tilde{\mathcal{A}}_k^{m+r} \subseteq \tilde{\mathcal{B}}_k^{m+r}$ via a natural embedding π which embeds the left-most copy of $\mathcal{A}_k^{m-1}$ on layer $m-1$ of $\tilde{\mathcal{A}}_k^{m+r}$ into the left-most copy of $\mathcal{B}_k^{m-1}$ on layer $m-1$ of $\tilde{\mathcal{B}}_k^{m+r}$ (see Figs. 2 and 3); but
  - none of the adjoined constants lie in the left-most copy of $\mathcal{A}_k^{m-1}$ on layer $m-1$ of $\tilde{\mathcal{A}}_k^{m+r}$ nor in the left-most copy of $\mathcal{B}_k^{m-1}$ on layer $m-1$ of $\tilde{\mathcal{B}}_k^{m+r}$.

Then

$$\tilde{\mathcal{A}}_k^{m+r} \models \rho \text{ if, and only if, } \tilde{\mathcal{B}}_k^{m+r} \models \rho.$$

Proof In the following proof, it is probably beneficial to visualize a variable of some program scheme taking a value from the domain of some structure as the placing of a

pebble, corresponding to the variable, on the domain element in question. Throughout the proof we adopt the nomenclature of the statement of the theorem.

We begin with two lemmas. In both of these lemmas, we think of the computations of the program scheme $\rho \in \mathrm{NPSS}(m)$, for m odd (on some input structure), so that tests in while instructions (which may themselves be program schemes of $\mathrm{NPSS}(m-1)$) are simply evaluated as either true or false. That is, we only consider such computations at the 'top level' and think of the computations as consisting of sequences of tuples of values of the input-output variables of ρ, together with some flow control, i.e., the number of the instruction about to be executed. Statements such as 'no input-output variable of ρ ever takes a value from the left-most copy of $\mathcal{A}_k^{m-2}$ on layer $m-2$ of $\tilde{\mathcal{A}}_k^{m+r}$' are intended to apply to this 'top-level' view of computations.

Lemma 14 If $m \geq 3$ is odd and $\tilde{\mathcal{A}}_k^{m+r} \models \rho$ then there is an accepting computation of ρ on $\tilde{\mathcal{A}}_k^{m+r}$ such that:

- 0 and max are taken as elements u and v such that $U_{m+r}(u)$ and $U_{m+r-1}(v)$ hold in $\tilde{\mathcal{A}}_k^{m+r}$, with v in the right-most copy of $\mathcal{A}_k^{m+r-1}$ or $\mathcal{B}_k^{m+r-1}$ on layer $m+r-1$ of $\tilde{\mathcal{A}}_k^{m+r}$;
- no input-output variable of ρ ever takes a value from the left-most copy of $\mathcal{A}_k^{m-2}$ on layer $m-2$ of $\tilde{\mathcal{A}}_k^{m+r}$.

Proof We shall simulate an accepting computation of ρ on $\tilde{\mathcal{A}}_k^{m+r}$ with an accepting computation satisfying the statement of the lemma. Choose 0 and max as in the statement of the lemma, and denote the expansion of $\tilde{\mathcal{A}}_k^{m+r}$ with the constants 0 and max as $(\tilde{\mathcal{A}}_k^{m+r}, 0, max)$ (and similarly for other tuples of values).

As stated prior to this lemma: ρ is essentially considered as a program scheme of NPSS(1) where tests in while instructions are simply evaluated as true or false; and computations are considered as sequences of tuples of values of the input-output variables, together with some flow control. Suppose that our original computation is the sequence $\{(\alpha_i, I_i)\}_{i=1}^{c}$, where each α_i is a tuple of values for the input-output variables and each I_i is the instruction about to be executed: so, $\alpha_1 = (0, 0, \ldots, 0)$ and $I_1 = 2$ (to denote the second instruction). What we do is to construct a new sequence $\{(\beta_i, I_i)\}_{i=1}^{c}$ so that this sequence corresponds to a proper computation of ρ on $\tilde{\mathcal{A}}_k^{m+r}$ and so that the two structures obtained as expansions of $(\tilde{\mathcal{A}}_k^{m+r}, 0, max)$ by adjoining constants corresponding to the values of α_i and β_i are isomorphic. Moreover, we shall ensure that no value of any β_i comes from the left-most copy of $\mathcal{A}_k^{m-2}$ on layer $m-2$ of $\tilde{\mathcal{A}}_k^{m+r}$.

Our construction of β_i proceeds by induction (the base case is when $\alpha_1 = (0, 0, \ldots, 0) = \beta_1$). Suppose that $(\tilde{\mathcal{A}}_k^{m+r}, 0, max, \alpha_i)$ is isomorphic to $(\tilde{\mathcal{A}}_k^{m+r}, 0, max, \beta_i)$ via the isomorphism θ_i. If instruction I_i is an assignment or a test evaluation then we are done (because whether any test is true or false is invariant under isomorphism, and none of the s adjoined constants, as in the statement of the main theorem, lie in the left-most copy of $\mathcal{A}_k^{m-1}$ on layer $m-1$ of $\tilde{\mathcal{A}}_k^{m+r}$). If instruction I_i is a guess instruction then there are two possibilities: the 'natural simulating guess', according to θ_i, is not in the left-most copy of $\mathcal{A}_k^{m-2}$ on layer $m-2$ of $\tilde{\mathcal{A}}_k^{m+r}$, or it is. In the first case, we are done. Alternatively, we can make our guess in a 'free copy' of $\mathcal{A}_k^{m-2}$, i.e., a copy in which no variable currently has a value, from amongst the $k+1$ copies

of $\mathcal{A}_k^{m-2}$ on layer $m-2$ of $\bar{\mathcal{A}}_k^{m+r}$ adjacent to the left-most copy (at least one of these $k+1$ copies is 'free'). Of course, the resulting structures $(\bar{\mathcal{A}}_k^{m+r}, 0, max, \alpha_{i+1})$ and $(\bar{\mathcal{A}}_k^{m+r}, 0, max, \beta_{i+1})$ are no longer isomorphic via θ_i but we can always ensure that they remain isomorphic via another isomorphism θ_{i+1}. The result follows.

Lemma 15 If $m \geq 1$ is odd and $\tilde{\mathcal{B}}_k^{m+r} \models \rho$ then there is an accepting computation of ρ on $\tilde{\mathcal{B}}_k^{m+r}$ such that:

- 0 and max are taken as elements u and v such that $U_{m+r}(u)$ and $U_{m+r-1}(v)$ hold in $\tilde{\mathcal{B}}_k^{m+r}$, with v in the right-most copy of $\mathcal{A}_k^{m+r-1}$ or $\mathcal{B}_k^{m+r-1}$ on layer $m+r-1$ of $\tilde{\mathcal{B}}_k^{m+r}$;
- if $m = 1$ then no input-output variable of ρ ever takes a value from the left-most copy of $\mathcal{B}^0$ on layer 0 of $\tilde{\mathcal{B}}_k^{m+r}$; and
- if $m \geq 3$ then no input-output variable of ρ ever takes a value from the left-most copy of $\mathcal{B}_k^{m-1}$ on layer $m-1$ of $\tilde{\mathcal{B}}_k^{m+r}$.

Proof Follows immediately if we argue as we did in the proof of Lemma 14.

We now proceed by induction on m.

Base Case Fix $m = 1$.
Suppose that $\tilde{\mathcal{A}}_k^{1+r} \models \rho$. Let 0 and max be elements u and v such that $U_{1+r}(u)$ and $U_r(v)$ hold in $\tilde{\mathcal{A}}_k^{1+r}$, with v in the right-most copy of $\mathcal{A}_k^r$ or $\mathcal{B}_k^r$ on layer r of $\tilde{\mathcal{A}}_k^{1+r}$. As $\tilde{\mathcal{A}}_k^{1+r} \subseteq \tilde{\mathcal{B}}_k^{1+r}$ via π, we can mirror an accepting computation of ρ on $\tilde{\mathcal{A}}_k^{1+r}$ in a computation of ρ on $\tilde{\mathcal{B}}_k^{1+r}$ (with 0 and max in the computation of ρ on $\tilde{\mathcal{B}}_k^{1+r}$ taken as the images of 0 and max in the computation of ρ on $\tilde{\mathcal{A}}_k^{1+r}$ under π) until we encounter a test evaluation. Let $\bar{\mathcal{A}}_k^{1+r}$ and $\bar{\mathcal{B}}_k^{1+r}$ be the expansions of $\tilde{\mathcal{A}}_k^{1+r}$ and $\tilde{\mathcal{B}}_k^{1+r}$, respectively, by adjoining additional constants whose values are the values of the input-output variables of ρ in the two computations. As we have been following π in our computation of ρ on $\tilde{\mathcal{B}}_k^{1+r}$, $\bar{\mathcal{A}}_k^{1+r} \subseteq \bar{\mathcal{B}}_k^{1+r}$ via π. As any test is a quantifier-free first-order formula φ, either φ is true in both $\bar{\mathcal{A}}_k^{1+r}$ and $\bar{\mathcal{B}}_k^{1+r}$ or φ is false in both $\bar{\mathcal{A}}_k^{1+r}$ and $\bar{\mathcal{B}}_k^{1+r}$. Thus, the flow of control in both computations of ρ is identical. By continuing the computation of ρ in $\tilde{\mathcal{B}}_k^{1+r}$ as dictated by π and arguing as above, we obtain that $\tilde{\mathcal{B}}_k^{1+r} \models \rho$.

Conversely, suppose that $\tilde{\mathcal{B}}_k^{1+r} \models \rho$. By Lemma 15, we can assume that in the accepting computation of ρ on $\tilde{\mathcal{B}}_k^{1+r}$: 0 and max are taken as elements u and v such that $U_{1+r}(u)$ and $U_r(v)$ hold in $\tilde{\mathcal{B}}_k^{1+r}$, with v in the right-most copy of $\mathcal{A}_k^r$ or $\mathcal{B}_k^r$ on layer r of $\tilde{\mathcal{B}}_k^{1+r}$; and no input-output variable ever takes a value from the left-most copy of $\mathcal{B}^0$ on layer 0 of $\tilde{\mathcal{B}}_k^{1+r}$. Given this fact, we can clearly mirror our accepting computation of ρ on $\tilde{\mathcal{B}}_k^{1+r}$ in a computation of ρ on $\tilde{\mathcal{A}}_k^{1+r}$ until we encounter a test evaluation for which the test is a quantifier-free first-order formula φ. Let $\bar{\mathcal{A}}_k^{1+r}$ and $\bar{\mathcal{B}}_k^{1+r}$ be the expansions of $\tilde{\mathcal{A}}_k^{1+r}$ and $\tilde{\mathcal{B}}_k^{1+r}$, respectively, by adjoining constants whose values are the values of the input-output variables of ρ immediately prior to the test evaluation encountered in both computations. Clearly, $\bar{\mathcal{A}}_k^{1+r} \subseteq \bar{\mathcal{B}}_k^{1+r}$ via π, and so either φ is true in both $\bar{\mathcal{A}}_k^{1+r}$ and $\bar{\mathcal{B}}_k^{1+r}$ or φ is false in both $\bar{\mathcal{A}}_k^{1+r}$ and $\bar{\mathcal{B}}_k^{1+r}$. Hence, as above, $\tilde{\mathcal{A}}_k^{1+r} \models \rho$.

Induction Step (a) The result holds for all m' such that $m' < m$, where $m \geq 3$ is odd and for all $r \geq 0$.

Suppose that $\tilde{\mathcal{A}}_k^{m+r} \models \rho$. By Lemma 14, we can assume that in the accepting computation of ρ on $\bar{\mathcal{A}}_k^{m+r}$: 0 and max are taken as elements u and v such that $U_{m+r}(u)$ and $U_{m+r-1}(v)$ hold in $\bar{\mathcal{A}}_k^{m+r}$, with v in the right-most copy of $\mathcal{A}_k^{m+r-1}$ or $\mathcal{B}_k^{m+r-1}$ on layer $m+r-1$ of $\bar{\mathcal{A}}_k^{m+r}$; and no input-output variable ever takes a value from the left-most copy of $\mathcal{A}_k^{m-2}$ on layer $m-2$ of $\bar{\mathcal{A}}_k^{m+r}$. As $\bar{\mathcal{A}}_k^{m+r} \subseteq \bar{\mathcal{B}}_k^{m+r}$ via π, we can mirror our accepting computation of ρ on $\bar{\mathcal{A}}_k^{m+r}$ in a computation of ρ on $\bar{\mathcal{B}}_k^{m+r}$ by using π until we encounter a test evaluation for which the test is a program scheme $\rho' \in \mathrm{NPS}(m-1)$. Note that this program scheme ρ' might have additional free variables to the free variables of ρ. Let $\bar{\mathcal{A}}_k^{m+r}$ and $\bar{\mathcal{B}}_k^{m+r}$ be the expansions of $\bar{\mathcal{A}}_k^{m+r}$ and $\bar{\mathcal{B}}_k^{m+r}$, respectively, by adjoining constants whose values are the values of the additional free variables of ρ' immediately prior to the test evaluation encountered in both computations. As our computation of ρ on $\bar{\mathcal{B}}_k^{m+r}$ has been proceeding according to π, we have that: $\bar{\mathcal{A}}_k^{m+r} \subseteq \bar{\mathcal{B}}_k^{m+r}$ via a natural embedding π which embeds the left-most copy of $\mathcal{A}_k^{m-2}$ on layer $m-2$ of $\bar{\mathcal{A}}_k^{m+r}$ into the left-most copy of $\mathcal{B}_k^{m-2}$ on layer $m-2$ of $\bar{\mathcal{B}}_k^{m+r}$; but where no constants lie in the left-most copy of $\mathcal{A}_k^{m-2}$ on layer $m-2$ of $\bar{\mathcal{A}}_k^{m+r}$ nor in the left-most copy of $\mathcal{B}_k^{m-2}$ on layer $m-2$ of $\bar{\mathcal{B}}_k^{m+r}$. Hence, by the induction hypothesis (with $m+r$ rewritten as $(m-1)+(r+1)$), $\bar{\mathcal{A}}_k^{m+r} \models \rho'$ if, and only if, $\bar{\mathcal{B}}_k^{m+r} \models \rho'$. Thus, either the test evaluation is true in both computations of ρ or false in both computations of ρ. Clearly, by continuing in this manner, the computation of ρ on $\bar{\mathcal{B}}_k^{m+r}$ is accepting.

Conversely, suppose that $\tilde{\mathcal{B}}_k^{m+r} \models \rho$. By Lemma 15, we can assume that in the accepting computation of ρ on $\bar{\mathcal{B}}_k^{m+r}$: 0 and max are taken as elements u and v such that $U_{m+r}(u)$ and $U_{m+r-1}(v)$ hold in $\tilde{\mathcal{B}}_k^{m+r}$, with v in the right-most copy of $\mathcal{A}_k^{m+r-1}$ or $\mathcal{B}_k^{m+r-1}$ on layer $m+r-1$ of $\tilde{\mathcal{B}}_k^{m+r}$; and no input-output variable ever takes a value from the left-most copy of $\mathcal{B}_k^{m-1}$ on layer $m-1$ of $\tilde{\mathcal{B}}_k^{m+r}$. Given this fact, we can clearly mirror our accepting computation of ρ on $\bar{\mathcal{B}}_k^{m+r}$ in a computation of ρ on $\bar{\mathcal{A}}_k^{m+r}$ until we encounter a test evaluation for which the test is a program scheme $\rho' \in \mathrm{NPS}(m-1)$. Let $\bar{\mathcal{A}}_k^{m+r}$ and $\bar{\mathcal{B}}_k^{m+r}$ be the expansions of $\bar{\mathcal{A}}_k^{m+r}$ and $\bar{\mathcal{B}}_k^{m+r}$, respectively, by adjoining constants whose values are the values of the free variables of ρ' immediately prior to the test evaluation encountered in both computations. Clearly: $\bar{\mathcal{A}}_k^{m+r} \subseteq \bar{\mathcal{B}}_k^{m+r}$ via a natural embedding π which embeds the left-most copy of $\mathcal{A}_k^{m-2}$ on layer $m-2$ of $\bar{\mathcal{A}}_k^{m+r}$ into the left-most copy of $\mathcal{B}_k^{m-2}$ on layer $m-2$ of $\bar{\mathcal{B}}_k^{m+r}$; but where no constants lie in the left-most copy of $\mathcal{A}_k^{m-2}$ on layer $m-2$ of $\bar{\mathcal{A}}_k^{m+r}$ nor in the left-most copy of $\mathcal{B}_k^{m-2}$ on layer $m-2$ of $\bar{\mathcal{B}}_k^{m+r}$. Hence, by the induction hypothesis (with $m+r$ rewritten as $(m-1)+(r+1)$), $\bar{\mathcal{A}}_k^{m+r} \models \rho'$ if, and only if, $\bar{\mathcal{B}}_k^{m+r} \models \rho'$. Consequently, as above, $\tilde{\mathcal{A}}_k^{m+r} \models \rho$.

Induction Step (b) The result holds for all m' such that $m' < m$ where $m \geq 2$ is even, and for all $r \geq 0$.

The program scheme ρ is of the form $\forall x_1 \forall x_2 \ldots \forall x_p \rho'$, for some p and for some $\rho' \in \mathrm{NPS}(m-1)$. Suppose that $\tilde{\mathcal{A}}_k^{m+r} \models \forall x_1 \forall x_2 \ldots \forall x_p \rho'$. Let $\bar{\mathcal{B}}_k^{m+r}$ be an expansion of $\tilde{\mathcal{B}}_k^{m+r}$ by adjoining p additional constants. We may assume that none of these additional p constants lie in the left-most copy of $\mathcal{B}_k^{m-2}$ on layer $m-2$ of $\bar{\mathcal{B}}_k^{m+r}$ (the

program scheme only involves k variables and so there will always be a 'free copy' of $\mathcal{B}_k^{m-2}$ on layer $m-2$ of $\bar{\mathcal{B}}_k^{m+r}$ which we may assume to be the left-most one). Let $\bar{\mathcal{A}}_k^{m+r}$ be the expansion of $\tilde{\mathcal{A}}_k^{m+r}$ with p additional constants such that $\bar{\mathcal{A}}_k^{m+r} \subseteq \bar{\mathcal{B}}_k^{m+r}$ via π. Note that no constant lies in the left-most copy of $\mathcal{A}_k^{m-2}$ on layer $m-2$ of $\bar{\mathcal{A}}_k^{m+r}$. As above, by the induction hypothesis, $\bar{\mathcal{A}}_k^{m+r} \models \rho'$ if, and only if, $\bar{\mathcal{B}}_k^{m+r} \models \rho'$. Thus, $\tilde{\mathcal{B}}_k^{m+r} \models \forall x_1 \forall x_2 \ldots \forall x_p \rho'$.

Conversely, suppose that $\tilde{\mathcal{B}}_k^{m+r} \models \forall x_1 \forall x_2 \ldots \forall x_p \rho'$. Let $\bar{\mathcal{A}}_k^{m+r}$ be an expansion of $\tilde{\mathcal{A}}_k^{m+r}$ by adjoining p additional constants. We may assume that none of these additional p constants lie in the left-most copy of $\mathcal{A}_k^{m-1}$ on layer $m-1$ of $\bar{\mathcal{A}}_k^{m+r}$. Let $\bar{\mathcal{B}}_k^{m+r}$ be the expansion of $\tilde{\mathcal{B}}_k^{m+r}$ with p additional constants such that $\bar{\mathcal{A}}_k^{m+r} \subseteq \bar{\mathcal{B}}_k^{m+r}$ via π. Note that no constant lies in the left-most copy of $\mathcal{B}_k^{m-1}$ on layer $m-1$ of $\bar{\mathcal{B}}_k^{m+r}$. As above, by the induction hypothesis, $\bar{\mathcal{A}}_k^{m+r} \models \rho'$ if, and only if, $\bar{\mathcal{B}}_k^{m+r} \models \rho'$. Thus, $\tilde{\mathcal{A}}_k^{m+r} \models \forall x_1 \forall x_2 \ldots \forall x_p \rho'$.

The result follows.

It is appropriate that we make a few remarks about the proof of Theorem 13. We give the proof in considerable detail for two reasons. First, our proof is for classes of program schemes as opposed to logics and is very different in nature to those in [13, 21] and [22] where the essential tools are Ehrenfeucht-Fraïssé games for transitive closure logic and bounded variable infinitary logic, respectively. As we soon see, hierarchy results similar to those from [13, 21, 22] follow as easy corollaries from our results. We make no mention whatsoever of any sort of games: we merely consider computations in program schemes. Second, we shall require our detailed proof later when we consider extended classes of program schemes.

Theorem 13 is the mechanism by which we tie the structures $\mathcal{A}_k^m$ and $\mathcal{B}_k^m$ together; the following proposition is used to pull them apart.

Proposition 16 Let $\mathcal{A}^0$ and $\mathcal{B}^0$ be $\langle E, U_0 \rangle$-structures, where E is a binary relation symbol and U_0 is a unary relation symbol, such that:

- $|\mathcal{A}^0| = 1$ and $|\mathcal{B}^0| = 2$;
- the unique element of $\mathcal{A}^0$ is in U_0 and one of the two elements of $\mathcal{B}^0$ is in U_0; and
- $E = \{(y,z),(z,y)\}$ in $\mathcal{B}^0$, where y and z are the two distinct elements.

For every $m \geq 1$, there exists a first-order sentence Ψ_m of the form

$$\forall x_{m-1} \exists x_{m-2} \ldots \exists x_1 \forall x_0 \exists y \psi, \text{ if } m \text{ is odd},$$

and

$$\exists x_{m-1} \forall x_{m-2} \ldots \exists x_1 \forall x_0 \exists y \psi, \text{ if } m \text{ is even},$$

where ψ is quantifier-free first-order, such that for every $k \geq 1$,

$$\mathcal{B}_k^m \models \Psi_m \text{ and } \mathcal{A}_k^m \not\models \Psi_m.$$

Proof In the following, 'x is an E-neighbour of y' means that both $E(x,y)$ and $E(y,x)$ hold.

Let $m \geq 1$ be odd. Consider the following first-order sentence Ψ_m, defined as: "For every $x_{m-1} \in U_{m-1}$, there exists an E-neighbour $x_{m-2} \in U_{m-2}$ of x_{m-1} such that for every E-neighbour $x_{m-3} \in U_{m-3}$ of x_{m-2}, there exists ... such that for every E-neighbour $x_0 \in U_0$ of x_1, there exists an E-neighbour y of x_0 that is not in U_1."

Let $m \geq 2$ be even. Consider the following first-order sentence Ψ_m, defined as: "There exists $x_{m-1} \in U_{m-1}$ such that for every E-neighbour $x_{m-2} \in U_{m-2}$ of x_{m-1}, there exists an E-neighbour $x_{m-3} \in U_{m-3}$ of x_{m-2} such that for every ... such that for every E-neighbour $x_0 \in U_0$ of x_1, there exists an E-neighbour y of x_0 that is not in U_1."

Clearly, Ψ_m is of the required form, and $\mathcal{B}_k^m \models \Psi_m$ and $\mathcal{A}_k^m \not\models \Psi_m$.

Let $\Sigma_0 = \Pi_0$ be the set of quantifier-free first-order formulae, and for each $m \geq 1$, let Σ_m (resp. Π_m) be the set of first-order formulae of the form

$$\exists x_1 \ldots \exists x_k \varphi \text{ (resp. } \forall x_1 \ldots \forall x_k \varphi),$$

where $\varphi \in \Pi_{m-1}$ (resp. $\varphi \in \Sigma_{m-1}$). We can now obtain our basic hierarchy theorem for the class of program schemes NPS.

Corollary 17 For every $m \geq 1$, there is a problem in Π_{m+1}, if m is odd, and Σ_{m+1}, if m is even, which is not in NPS(m). Consequently,

$$\text{NPS}(1) \subset \ldots \subset \text{NPS}(m) \subset \text{NPS}(m+1) \subset \ldots$$

Proof Let $\mathcal{A}^0$ and $\mathcal{B}^0$ be the structures over the signature $\sigma = \langle E, U_0 \rangle$ as in the statement of Proposition 16. Fix $m \geq 1$ and let $\rho \in \text{NPS}(m)$ be over the signature σ^m and involve k variables, none of which are free. By Theorem 13, $\mathcal{A}_k^m \models \rho$ if, and only if, $\mathcal{B}_k^m \models \rho$. However, the problem Ω_m defined by the sentence Ψ_m in the statement of Proposition 16, which clearly can be accepted by some program scheme of NPS($m+1$), is such that $\mathcal{B}_k^m \in \Omega_m$ and $\mathcal{A}_k^m \notin \Omega_m$.

Note from the proof of Corollary 17 that the problem separating NPS(m) from NPS(m+1) is over a signature determined by m. One might ask whether the hierarchy in Corollary 17 remains strict when we restrict ourselves to problems over some specific signature. In order to answer this question, let us focus on the proofs of Theorem 13 and Proposition 16, the essential results used to yield Corollary 17. The purpose of the relations $U_0, U_1, \ldots, U_m$ in the structures $\mathcal{A}_k^m$ and $\mathcal{B}_k^m$ is really just to add clarity and they can actually be dispensed with. Certainly, the proof of Theorem 13 goes through if we remove these relations from our structures. Now instantiate $\mathcal{A}^0$ and $\mathcal{B}^0$ as the graphs defined in Proposition 16. For every $m \geq 1$, we can easily build a sentence of the same quantifier structure as the sentence Ψ_m of Proposition 16 (call it Ψ_m also) and which tells the structures $\mathcal{A}_k^m$ and $\mathcal{B}_k^m$ apart when $\mathcal{A}_k^m$ and $\mathcal{B}_k^m$ are regarded as undirected trees in which the element v for which $U_m(v)$ formerly held is a distinguished root. Note that the root v is the middle vertex on a path in both $\mathcal{A}_k^m$ and $\mathcal{B}_k^m$ of length $2m+2$; and, indeed, it is the middle vertex in every path of length $2m+2$ in both $\mathcal{A}_k^m$ and $\mathcal{B}_k^m$. We can thus define the root v (without regarding it as distinguished) in both $\mathcal{A}_k^m$ and $\mathcal{B}_k^m$ using either a purely existential first-order

formula or a purely universal first-order formula (which is independent of k). By 'attaching' this universal or existential formula on to the front of our new sentence Ψ_m, if m is odd or even, respectively, we obtain a first-order sentence with the same quantifier-alternation pattern as the sentence Ψ_m in Proposition 16, and with the same properties. Also, by slightly amending the definition of the relation E in the structures $\mathcal{A}_k^m$ and $\mathcal{B}_k^m$ in the previous section, we can also consider $\mathcal{A}_k^m$ and $\mathcal{B}_k^m$ to be out-trees, i.e., trees where every edge is directed away from the root, and arrive at a similar conclusion. Hence, by arguing as in the proof of Corollary 17, we obtain the following result.

Corollary 18 For every $m \geq 1$, there is a problem in Π_{m+1}, if m is odd, and Σ_{m+1}, if m is even, which is not in NPS(m). Consequently,

$$\text{NPS}(1) \subset \ldots \subset \text{NPS}(m) \subset \text{NPS}(m+1) \subset \ldots$$

Moreover, the above results hold even when we only consider problems involving undirected trees or problems involving out-trees.

By a 'problem involving undirected trees' we mean that we restrict our domain of allowable structures just to σ_2-structures where the relation E is symmetric and where, when visualised as an undirected graph, these structures are of the form of a tree; and we define our problems to be (isomorphism-closed) subclasses of this domain. A 'problem involving out-trees' is defined similarly.

We can now use Theorem 13 and Proposition 16 to obtain some hierarchy results concerning different logics, similar or identical to already established results from the literature. There are also other immediate applications of Theorem 13 and Proposition 16: the results highlighted below merely serve to illustrate this fact. We reiterate that the proofs of the corollaries below do not involve Ehrenfeucht-Fraïssé games, unlike the proofs of these results from the literature.

In [7], Chandra and Harel showed (amongst other things) that the classes of problems defined by restricting the quantifier prefixes of first-order sentences in prenex normal form according to the number of alternations between universal and existential quantifiers form a proper hierarchy. This result is stated precisely below and its proof follows identically to those of Corollaries 17 and 18.

Corollary 19 [7]

$$\Sigma_1 \subset \Pi_2 \subset \Sigma_3 \subset \ldots,$$

even when we only consider problems involving undirected trees or problems involving out-trees.

Definition 20 Define:

- TC(1) to be the set of formulae of the form

  $$\text{TC}[\lambda \mathbf{x}, \mathbf{y} \psi](\mathbf{u}, \mathbf{v}),$$

  where ψ is quantifier-free first-order and where $\mathbf{u}$ and $\mathbf{v}$ are tuples of constant symbols or variables;

- TC$(m+1)$, for odd $m \geq 1$, to be the universal closure of TC(m), i.e., the set of formulae of the form $\forall z_1 \ldots \forall z_k \psi$, where ψ is a formula of TC(m); and
- TC$(m+1)$, for even $m \geq 2$, to be the set of formulae of the form

$$\mathrm{TC}[\lambda \mathbf{x}, \mathbf{y} \psi](\mathbf{u}, \mathbf{v}),$$

where $\psi \in \mathrm{TC}(m)$ and where $\mathbf{u}$ and $\mathbf{v}$ are tuples of constant symbols or variables (not necessarily distinct).

It is not difficult to show that TC(m) is closed under $\vee$ and $\wedge$, for each $m \geq 1$, and that $\cup\{\mathrm{TC}(m) : m \geq 1\} = \mathrm{TC}^*[\mathrm{FO}]$. (These fragments of transitive closure logic are very similar to those of the same name defined in [21]. Where Grädel's hierarchy differs from ours is that Grädel's base logic, which he called TC(0) but we call TC(1), was defined as ours is except that ψ was allowed to be *any* first-order formula and not just a quantifier-free one.)

The proof of the following is immediate from Theorems 11 and 13 and Proposition 16.

Corollary 21 In the presence of two built-in constants, for every $m \geq 1$, there is a problem in Π_{m+1}, if m is odd, and Σ_{m+1}, if m is even, which can not be defined by any sentence of $\pm\mathrm{TC}(m)$. In particular,

$$\mathrm{TC}(1) \subset \ldots \subset \mathrm{TC}(m) \subset \mathrm{TC}(m+1) \subset \ldots \ldots \subset \cup\{\mathrm{TC}(m) : m \geq 1\} = \mathrm{TC}^*[\mathrm{FO}]$$

and

$$\pm\mathrm{TC}(1) \subset \ldots \subset \pm\mathrm{TC}(m) \subset \pm\mathrm{TC}(m+1) \subset \ldots \ldots \subset \cup\{\pm\mathrm{TC}(m) : m \geq 1\} = (\pm\mathrm{TC})^*[\mathrm{FO}].$$

Moreover, the above results hold even when we only consider problems involving undirected trees or problems involving out-trees.

Let us take a diversion from our main path for a moment. Obviously, the reason that TC(m), for $m \geq 1$, is defined as it is, stems from the analogous definition of $\pm\mathrm{TC}(m)$ given earlier; and the reason that $\pm\mathrm{TC}(m)$ is defined as it is, stems from the alternative realisation of $\pm\mathrm{TC}(m)$ as NPS(m) (see Theorem 11). We could have allowed the class of program schemes NPS(1) to have first-order tests in their while instructions, and then built NPS(2), NPS(3), and so on, as before: the derived fragments of transitive closure logic corresponding to these new classes would then be identical to the fragments defined by Grädel in [21]. We could then have proven an amended version of Theorem 13 with extra stipulations on the structures $\mathcal{A}^0$ and $\mathcal{B}^0$ that they could not be distinguished by an appropriate first-order Ehrenfeucht-Fraïssé game; and used an amended version of Proposition 16 where $\mathcal{A}^0$ (resp. $\mathcal{B}^0$) is taken to consist of an appropriately large cycle (resp. a disjoint pair of appropriately large cycles) together with a disjoint vertex, the vertex of U_0, joined by an edge to every vertex of the cycle (resp. cycles) (these are the graphs used in [21]). Consequently, we can also obtain Grädel's exact hierarchy result (in fact, from our earlier discussion, we

can actually obtain Grädel's hierarchy result even on the class of undirected graphs; though not, of course, on the class of undirected trees).

Let us compare our hierarchy result with that of Grädel. Our hierarchy result is more refined (and we believe more interesting) than Grädel's for the following reason: we provide infinite hierarchies within $(\pm\mathrm{TC})^*[\mathrm{FO}]$ and $\mathrm{TC}^*[\mathrm{FO}]$ with the property that given any level of either of the hierarchies, there are *first-order definable* problems which are not in the given level, yet the union of all levels of the hierarchies gives $(\pm\mathrm{TC})^*[\mathrm{FO}]$ or $\mathrm{TC}^*[\mathrm{FO}]$, respectively. Also, in order for us to establish Grädel's hierarchy result, we would have had to play an Ehrenfeucht-Fraïssé game (albeit a first-order game and not a transitive closure game, as Grädel played) which is, in some sense, against the spirit of this paper. So ends our diversion (although we shall return to the possibility of establishing our hierarchy results using a result due to Grädel and McColm [22] later).

Our consideration of classes of program schemes as opposed to logics obviates the need to formalize and play games on structures: we simply mimic computation traces, and this is pedagogically clearer than using Ehrenfeucht-Fraïssé games. Also, focussing on program schemes as opposed to transitive closure logic (and its associated games) encourages us to develop other applications of our general approach, as we show now and, more importantly, in the next section.

For any problem Ω and for any $m \geq 1$, let the fragments $\Omega(m)$ and $\pm\Omega(m)$ of the logic $(\pm\Omega)^*[\mathrm{FO}]$ be defined analogously to $\mathrm{TC}(m)$ and $\pm\mathrm{TC}(m)$, respectively.

Let the problem CYC, over the signature σ_2, be defined as

$$\{\mathcal{A} \in \mathrm{STRUCT}(\sigma_2) : \text{ the graph } \mathcal{A} \text{ is cyclic}\}.$$

It is easy to show that $\mathrm{CYC} \in \mathrm{NPS}(1)$.

Corollary 22 For each $m \geq 1$, there are problems in Π_{m+1}, if m is odd, and Σ_{m+1}, if m is even, which can not be defined by any sentence of $\pm\mathrm{CYC}(m)$. In particular,

$$\mathrm{CYC}(1) \subset \ldots \subset \mathrm{CYC}(m) \subset \mathrm{CYC}(m+1) \subset \ldots \\ \ldots \subset \cup\{\mathrm{CYC}(m) : m \geq 1\} = \mathrm{CYC}^*[\mathrm{FO}|$$

and

$$\pm\mathrm{CYC}(1) \subset \ldots \subset \pm\mathrm{CYC}(m) \subset \pm\mathrm{CYC}(m+1) \subset \ldots \\ \ldots \subset \cup\{\pm\mathrm{CYC}(m) : m \geq 1\} = (\pm\mathrm{CYC})^*[\mathrm{FO}].$$

Moreover, the above results hold even when we only consider problems involving undirected trees or problems involving out-trees.

Proof Consider the sentence $\exists x\theta(x)$, where $\theta(x) \in (\pm\mathrm{CYC})^*[\mathrm{FO}]$. Define the formula $\psi(x_1, x_2, x_3, y_1, y_2, y_3)$ as

$$(x_1 = y_2 \neq y_1 = x_2 \wedge x_3 = y_3 \wedge \theta(x_3)) \vee (x_1 = x_2 = y_1 \neq x_3 = y_2) \\ \vee (x_1 = x_2 = y_3 \neq x_3 = y_1 = y_2).$$

When we interpret ψ in some appropriate structure $\mathcal{A}$ (of size at least 2), we obtain a graph G whose vertex set is $|\mathcal{A}|^3$ and whose edge set is

$$\{(\mathbf{u}, \mathbf{v}) \in |\mathcal{A}|^3 : \psi(\mathbf{u}, \mathbf{v}) \text{ or } \psi(\mathbf{v}, \mathbf{u}) \text{ holds in } \mathcal{A}\}.$$

Define $V_{u,v} = V_{v,u} = \{(u,v,w),(v,u,w),(u,u,v),(v,v,u) : w \in |\mathcal{A}|\}$, for distinct u and v in $|\mathcal{A}|$. These sets of vertices are disjoint. Then every edge of G has both its endpoints in some $V_{u,v}$; and so G has a cycle if, and only if, the subgraph of G induced by the vertices of some $V_{u,v}$ has a cycle. But, all $V_{u,v}$'s are isomorphic and any $V_{u,v}$ has a cycle if, and only if, $\mathcal{A} \models \exists x\theta(x)$. Hence, for every $m \geq 1$, it is not difficult to see that the sentence Ψ_m of Proposition 16 is logically equivalent to a sentence of CYC$(m+1)$.

Clearly, CYC $\in$ NPS(1); and so every problem of $\pm$CYC(m) is accepted by some program scheme of NPS(m), for each $m \geq 1$. Thus, as a simple induction yields that $\cup\{\text{CYC}(m) : m \geq 1\} = \text{CYC}^*[\text{FO}]$ and $\cup\{\pm\text{CYC}(m) : m \geq 1\} = (\pm\text{CYC})^*[\text{FO}]$, the result follows from Theorem 13 and Proposition 16.

Note that we used the fact that CYC is in NPS(1) to obtain Corollary 22. We need not be so severe (although the resulting hierarchy results are not as satisfactory as those just mentioned). Let ROOT be the problem detailed in Example 9; that is, ROOT consists of all those structures over the signature $\sigma = \langle E, U\rangle$, where E is a binary relation symbol and U is a unary relation symbol, such that when these structures are considered as rooted digraphs (with the roots given by U), at least one of the roots is such that there are paths from this root to every other vertex.

Corollary 23 In the presence of two built-in constants, for each even $m \geq 1$, there are problems in Σ_{m+1} which can not be defined by any sentence of $\pm$ROOT(m). Hence, the hierarchies

$$\text{ROOT}(1) \subseteq \ldots \subseteq \text{ROOT}(m) \subseteq \text{ROOT}(m+1) \subseteq \ldots$$

and

$$\pm\text{ROOT}(1) \subseteq \ldots \subseteq \pm\text{ROOT}(m) \subseteq \pm\text{ROOT}(m+1) \subseteq \ldots$$

do not collapse. Moreover, the above results hold even when we only consider problems involving undirected trees or problems involving out-trees.

Proof Consider the formula $\exists x\theta(x)$, where $\theta \in (\pm\text{ROOT})^*[\text{FO}]$. Then $\exists x\theta(x)$ is equivalent to the formula

$$\begin{aligned}\text{ROOT}[\lambda(x_1,x_2)(x_1 = 0 \wedge x_2 = 0),(y_1,y_2),(z_1,z_2)((y_1 = 0 \wedge y_2 = 0 \\ \wedge(z_1 \neq 0 \vee z_2 \neq max)) \vee (y_1 = max \wedge \theta(y_2) \wedge z_1 = 0 \wedge z_2 = max))],\end{aligned}$$

where x_1, x_2, y_1, y_2, z_1 and z_2 are new variables. Hence, for every $m \geq 1$, it is not difficult to see that the sentence Ψ_m of Proposition 16 is logically equivalent to a sentence of ROOT$(m+1)$.

By Example 9, the problem ROOT is accepted by some program scheme of NPS(3). Thus, a simple induction yields that the problem accepted by a sentence of $\pm$ROOT(m) can be accepted by a program scheme of NPS$(2m)$, if $m \geq 2$ is even, and NPS$(2m+1)$, if $m \geq 1$ is odd. Let $m \geq 2$ be even. By Theorem 13 and Proposition 16, there are problems definable in ROOT$(2m+1)$ which are not definable in $\pm$ROOT(m), and the result follows.

Let co-2SAT be the problem over $\sigma_{2,2}$ defined as all those $\sigma_{2,2}$-structures such that when considered as a collection of clauses, they form an unsatisfiable collection in which every clause has 0 or 2 distinct literals.

Corollary 24 For each even $m \geq 1$, there are problems in Σ_{m+1} which can not be defined by any sentence of $\pm$co-2SAT(m). Hence, the hierarchies

$$\text{co-2SAT}(1) \subseteq \ldots \subseteq \text{co-2SAT}(m) \subseteq \text{co-2SAT}(m+1) \subseteq \ldots$$

and

$$\pm\text{co-2SAT}(1) \subseteq \ldots \subseteq \pm\text{co-2SAT}(m) \subseteq \pm\text{co-2SAT}(m+1) \subseteq \ldots$$

do not collapse. Moreover, the above results hold even when we only consider problems involving undirected trees or problems involving out-trees.

Proof Consider the formula $\exists x\theta(x)$, where $\theta(x) \in (\pm\text{co-2SAT})^*[\text{FO}]$. Then $\exists x\theta(x)$ is logically equivalent to the formula

$$\begin{aligned}\text{co-2SAT}[\lambda(x_1, y_1), (x_2, y_2)(\theta(x_1) \wedge x_1 = y_1 = x_2 = y_2),\\ (x_1, y_1), (x_2, y_2)(\theta(x_1) \wedge x_1 = y_1 \wedge x_2 \neq y_2)],\end{aligned}$$

where x_1, y_1, x_2 and y_2 are new variables. Hence, for every $m \geq 1$, it is not difficult to see that the sentence Ψ_m of Proposition 16 is logically equivalent to a sentence of co-2SAT($m+1$).

By Example 4, the problem co-2SAT is accepted by some program scheme of NPS(3) (as checking to see that every clause has 0 or 2 distinct literals can be done by a program scheme of NPS(3)). The result follows similarly to as in Corollary 23.

6 Extending program schemes with a stack

Clearly, many other hierarchies similar to those in Corollaries 21, 22 and 23 can be obtained by proceeding as we did in the proofs of these corollaries; and it is well worth attempting to establish necessary and sufficient conditions on problems, such as TC, CYC and ROOT, for such hierarchies to exist. What is apparent is that if we are to obtain logical hierarchies by proceeding in this way then a necessary condition on any corresponding problem (such as TC, CYC or ROOT) is that it is in the complexity class **NL** (by Theorem 12). One of the main contributions in this paper is a means by which we can establish such logical hierarchies where the corresponding problem is probably not in **NL** (following some widely accepted complexity-theoretic beliefs), and it is here that we turn now.

So far, we have, essentially, replicated and refined some results from the literature which had hitherto been proven using Ehrenfeucht-Fraïssé games, and not by considering program scheme computations as we do here. Our shift in focus from logics and games to program schemes and computations enables us to enhance our program schemes by a means not available to us in the logical setting; namely, we can add a stack to our program schemes.

Definition 25 For any $m \geq 1$, a program scheme of NPSS(m) is defined exactly as was a program scheme of NPS(m) except that there are two additional instructions:

- $x_i :=$ POP; and
- PUSH x_i.

The new instructions provide access to a stack in the usual way. That is: when the instruction 'PUSH x_i' is encountered in some program scheme, the value of the variable x_i is placed on the top of the stack (so increasing the height of the stack by 1) but so that x_i retains its value; and when the instruction '$x_i :=$ POP' is encountered, the value on the top of the stack is removed (so decreasing the height of the stack by 1) and the variable x_i assumes this value. Note that there is no test to see whether the stack is empty. However, extra clarification is in order.

Let ρ be a program scheme of NPSS(m), for some odd $m \geq 1$. A computation on some input structure proceeds as usual, starting with an empty stack, until a test evaluation is encountered (note that if ever an instruction '$x_i :=$ POP' is encountered in some computation when the stack is empty then the computation 'hangs', i.e., does not terminate). The test involves a program scheme of NPSS($m-1$) of the form $\forall x_1 \forall x_2 \ldots \forall x_p \rho'$, for some program scheme ρ' of NPSS($m-2$). Upon encountering this test evaluation, the stack of ρ remains fixed until the truth or falsity of the test has been established. In order to establish the truth or falsity of the test, as before we consider computations of the program scheme ρ', one for each possible valuation of the variables $x_1, x_2, \ldots x_p$. In each of these computations, ρ' starts with an empty stack. Hence, any computation of a program scheme has its own associated stack. Having established whether the test evaluation results in true or false, the computation of ρ resumes accordingly. Computations of program schemes of NPSS(m) for even $m \geq 2$ are defined similarly.

Remark 26 Even though we have no test to see whether a stack is empty or not, we can always assume that an input is accepted by some program scheme of NPSS(m) if, and only if, it is accepted such that on termination the stack is empty. We do this by simulating our original program scheme, with another program scheme of NPSS(m), as follows. We simulate a push in our original program scheme by pushing first 0 and then the element in question onto the stack in our simulating program scheme, with a pop simulated by popping two elements from the stack. This allows us to have a unique 'bottom element', the pair of elements max and max, in our simulating program scheme which we initially push onto the stack. If ever the simulation is such that max and max are popped then:

- if the original program scheme has accepted at this point then we accept in our simulation (with an empty stack); and
- if the original program scheme has not accepted at this point (and so is trying to pop from an empty stack) then we reject in our simulation.

Also, if our original program scheme has accepted then in our simulation we pop everything off the stack (that is, until we have popped the pair max and max) and accept.

Example 27 Consider the following program scheme of NPSS(1) over the signature σ_{3++}.

1. INPUT$(x_1, x_2, x_3, x_4, x_5)$
2. PUSH C
3. PUSH C

4. WHILE $x_1 = 0$ DO
5. GUESS x_2
6. $x_3 :=$ POP
7. $x_4 :=$ POP
8. GUESS x_5
9. IF $x_5 = 0$ THEN
10. PUSH x_3 FI
11. GUESS x_5
12. IF $x_5 = 0$ THEN
13. PUSH x_4 FI
14. IF $R(C, x_3, x_2) \vee R(x_3, C, x_2) \vee R(C, x_4, x_2) \vee R(x_4, C, x_2)$
$\vee R(x_3, x_4, x_2) \vee R(x_4, x_3, x_2) \vee R(C, C, x_2) \vee R(x_3, x_3, x_2)$
$\vee R(x_4, x_4, x_2) \vee x_2 = C$ THEN
15. GUESS x_5
16. IF $x_5 = 0$ THEN
17. PUSH x_2 FI
18. IF $x_2 = D$ THEN
19. $(x_1, x_2, x_3, x_4, x_5) := (max, max, max, max, max)$ FI FI OD
20. OUTPUT$(x_1, x_2, x_3, x_4, x_5)$

Suppose that the σ_{3++}-structure $\mathcal{A}$ is in PS; that is, the vertex D is accessible from the vertex C. Let the vertices of $\{C = C_0, C_1, \ldots, C_a = D\}$ be accessible where for every $j \geq 1$, C_j can be shown to be accessible by applying a rule $(C_{j_1}, C_{j_2}) \mapsto C_j$ where both j_1 and j_2 are less than j. Let the following be our induction hypothesis IH(i), where $i < a$:

- for any two elements of $\{C_0, C_1, \ldots, C_i\}$, there exists a computation of ρ on $\mathcal{A}$ such that both these elements are on the stack at the same time and at this time the flow of control in ρ is ready to execute the while instruction.

Trivially, IH(0) and IH(1) hold.

Suppose that IH(i) holds for some $i < a - 1$, as does the rule $(C_{j_1}, C_{j_2}) \mapsto C_{i+1}$, where both j_1 and j_2 are at most i. By IH(i), there exists a computation of ρ on $\mathcal{A}$, call it $comp(j_1, j_2)$, resulting in a configuration such that both C_{j_1} and C_{j_2} are on the stack and the flow of control in ρ is at the while instruction. Clearly, there is an extension of this computation $comp(j_1, j_2)$ so that C_{j_1} and C_{j_2} are the top two items of the stack and the flow of control is at the while instruction (simply perform some more iterations of the while loop so that stack elements, apart from C_{j_1} and C_{j_2}, are 'thrown away').

Consider the subsequent computation which:

- guesses the value C_{i+1} for x_2 and then pushes C_{i+1} onto the stack, before returning the flow of control to the while instruction;
- guesses the value C for x_2 and then pushes C onto the stack, before returning the flow of control to the while instruction;
- guesses the value C for x_2 and then pushes C onto the stack, before returning the flow of control to the while instruction (so now the top two elements of the stack are C); and

- repeats the computation $comp(j_1, j_2)$, for *any* chosen j_1 and j_2 that are both at most i, before returning flow of control to the while instruction.

Note that C_{i+1}, C_{j_1} and C_{j_2} are now all on the stack. Hence, IH$(i+1)$ now holds.

So, by induction, IH(i) holds for all $i < a$. In particular, IH$(a-1)$ holds, and similar reasoning yields that there is a subsequent computation which pushes D onto the stack before forcing the while loop to terminate and thus causing $\mathcal{A}$ to be accepted by ρ.

Conversely, suppose that $\mathcal{A}$ is accepted by ρ. In order for a computation to be accepting, the variable x_2 must assume the value D and D must be yielded via a rule of the form $(x, y) \mapsto D$ where the elements x and y were previously (and might still be) on the stack. But only accessible vertices of $\mathcal{A}$ are ever placed on the stack, and so $\mathcal{A} \in$ PS. Consequently, the problem PS is in NPSS(1).

Example 27 shows that PS $\in$ NPSS(1). In fact, there is an even closer relationship between PS and the program schemes of NPSS(1), as we now demonstrate. But first, we need to introduce some terminology. This terminology is strongly influenced by [9] where it was shown that a non-deterministic pushdown automaton can be simulated by a deterministic pushdown automaton, and the class of languages (i.e., sets of strings over $\{0,1\}$) accepted by non-deterministic pushdown automata is **P**.

Definition 28 An *instantaneous description* (*ID*) of a k-variable program scheme $\rho \in$ NPSS(1) on some input structure consists of the label of the instruction of ρ about to be executed in the computation together with either a $(k+1)$-tuple detailing the values of the variables and the element on the top of the stack at that time or a k-tuple detailing the values of the variables if the stack is empty. The *values* of an ID are the values appearing in this tuple. If α and β are IDs then we write $\alpha = \beta$ to denote the facts that the instruction of ρ associated with α is identical to that associated with β and that the values of α and β are identical (as tuples).

Note that an ID does not necessarily describe the whole of the stack at some point in a computation of ρ on some input, just the top element, if it exists. Alternatively, we refer to a description of: the instruction about to be executed; the values of the variables; and the *whole* of the stack, at some point in a computation of ρ on this input, as a *configuration*. We can regard an ID α as a configuration $\bar{\alpha}$ by taking the stack to consist solely of the stack element of the ID, if there is one; and we can define ID of a configuration as the ID obtained from the configuration by ignoring everything below the top element of the stack, if there is one.

When we talk about some computation of some program scheme of NPSS(1) on some input structure $\mathcal{A}$, we assume that the constant symbols 0 and *max* have been fixed as some pair of distinct elements of $|\mathcal{A}|$.

Definition 29 Let (α_1, β_1) and (α_2, β_2) be two pairs of IDs of some program scheme $\rho \in$ NPSS(1) on some input structure. Then (α_1, β_1) and (α_2, β_2) *yield* the pair of IDs (α_3, β_3) if $\alpha_1 = \alpha_3$ and one of the following holds:

(a) either α_3 and β_3 have the same stack element or neither has a stack element; starting in configuration $\bar{\beta}_1$, it is possible for ρ to execute a push instruction and thus be in a configuration whose ID is α_2; and starting in configuration $\bar{\beta}_2$,

it is possible for ρ to execute a pop instruction and thus be in a configuration whose ID is the first k components of β_3 (that is, minus β_3's stack element, if it has one); or

(b) $\beta_1 = \alpha_2$ and either $\beta_2 = \beta_3$ or starting in configuration $\bar{\beta}_2$, it is possible for ρ to execute an instruction which is neither a push nor a pop instruction and thus be in the configuration $\bar{\beta}_3$.

If a pair of IDs (α, β) is eventually obtained by starting from a set Γ of pairs of IDs and continually applying the above yield rules then we say that (α, β) has been obtained by *applying the yield rules* to Γ.

Definition 30 Let (α, β) be a pair of IDs of some program scheme $\rho \in$ NPSS(1) on some input structure. Then (α, β) is *realizable* if:

- there is a (partial) computation of ρ on the input structure starting from the configuration $\bar{\alpha}$ and ending in the configuration $\bar{\beta}$ such that throughout this computation, the initial bottom stack element (that is, the stack element of α), if there is one, is never popped; and
- the ID α has a stack element if, and only if, the ID β has a stack element.

We can now prove our first property of accepting computations of program schemes of NPSS(1). Again, we are strongly influenced by [9].

Proposition 31 Every realizable pair (α, β) of IDs of some program scheme ρ of NPSS(1) where the input structure is $\mathcal{A}$ can be obtained from the set of all pairs of IDs of the form (γ, γ) by applying the yield rules.

Proof Suppose that the pair of IDs (α, β) is realizable. Then there is a computation of ρ on input $\mathcal{A}$ starting from the configuration $\bar{\alpha}$ and ending in the configuration $\bar{\beta}$, and so that this computation is as detailed in Definition 30. Let the length of this computation be t; that is, t is the number of instruction executions, or moves, to get from the configuration $\bar{\alpha}$ to the configuration $\bar{\beta}$. We shall prove by induction on t that (α, β) is as stated in the proposition. Let Γ denote the set of all pairs of IDs of the form (γ, γ).

If $t = 1$ then as (α, β) is realizable, the move taking the configuration $\bar{\alpha}$ to the configuration $\bar{\beta}$ can not be via a pop or a push instruction. So, we have that (α, α) and (α, α) yield (α, β). Suppose that the result holds for all computations of length t, and that there is a computation of length $t + 1$ taking the configuration $\bar{\alpha}$ to the configuration $\bar{\beta}$; moreover, suppose that this computation satisfies the conditions of Definition 30. Denote this computation by $\bar{\alpha}, c_1, c_2, \ldots, c_t, \bar{\beta}$ (note that this is a sequence of configurations, not IDs). There are two cases: the move taking the configuration c_t to $\bar{\beta}$ is via neither a pop nor a push instruction, or it is.

In the first case, the configuration c_t is of the form $\bar{\gamma}$, for some ID γ; and the pairs of IDs (α, γ) and (γ, β) are realizable. So, the induction hypothesis yields that (α, γ) and (γ, β) can be obtained from Γ by applying the yield rules. As (α, γ) and (γ, β) yield (α, β), we are done.

In the second case, the move taking the configuration c_t to $\bar{\beta}$ must be via a pop instruction; so, let a be the element at the top of the stack in the configuration c_t.

Let i be such that: the element at the top of the stack of c_i is a; the height of the stack of c_i is one more than the height of the stack of $\bar{\beta}$; the heights of the stacks of $c_{i+1}, c_{i+2}, \ldots, c_t$ are all at least the height of the stack of c_i; and the height of the stack of c_{i-1} is equal to the height of the stack of $\bar{\beta}$. That is, c_i is where the element a has been pushed onto the stack before it is popped off at c_t. Clearly, such an i exists and $1 \leq i \leq t$. Let γ, γ' and γ'' be the IDs obtained from the configurations c_{i-1}, c_i and c_t, respectively, by ignoring all elements of the stack except the top element, if there is one. Both pairs of IDs (α, γ) and (γ', γ'') are realizable, and so by the induction hypothesis they can be obtained from Γ by applying the yield rules. But (α, γ) and (γ', γ'') yield (α, β), and so (α, β) can be obtained from Γ by applying the yield rules. The result follows by induction.

We can now use Proposition 31 to tie together the logic $(\pm\mathrm{PS})^*[\mathrm{FO}]$ and the class of program schemes $\cup\{\mathrm{NPSS}(m) : m \geq 1\}$, which we denote by NPSS.

Theorem 32 In the presence of two built-in constant symbols, for each $m \geq 1$, $\pm\mathrm{PS}(m) = \mathrm{NPSS}(m)$; and consequently $(\pm\mathrm{PS})^*[\mathrm{FO}] = \mathrm{NPSS}$ (even in the absence of the two built-in constants).

Proof First, it is easy to show that any problem in $(\pm\mathrm{PS})^*[\mathrm{FO}]$ must be in $\pm\mathrm{PS}(m)$, for some $m \geq 1$. By Example 27, the problem PS is in NPSS(1), and consequently $\pm\mathrm{PS}(m) \subseteq \mathrm{NPSS}(m)$, for each $m \geq 1$.

Conversely, suppose that the problem Ω is accepted by some program scheme $\rho \in \mathrm{NPSS}(1)$ involving k variables. Let the structure $\mathcal{A}$ be accepted by ρ. By Remark 26, we may assume that the accepting computation of ρ on $\mathcal{A}$ is such that the final configuration has an empty stack. Hence, if α_0 is the unique initial ID and β_0 is the unique accepting ID then an input is accepted by ρ if, and only if, the pair of IDs (α_0, β_0) is realizable.

Any ID α of ρ on input $\mathcal{A}$ can be encoded by a tuple $\nu(\alpha)$ of length $k+l+2$, where l is the number of instructions in ρ. If the instruction of ρ associated with α is the ith, say, then each of the first l components of the tuple $\nu(\alpha)$ are 0 except the ith which is max; the next k components of $\nu(\alpha)$ consist of the values of the variables of α; and the last 2 components encode the stack element, if there is one, or the fact that there is no stack element. Hence, a pair of IDs (α, β) can be encoded by the concatenation $(\nu(\alpha), \nu(\beta))$ of the two corresponding tuples. Also, there is clearly a quantifier-free first-order formula which ascertains whether a $2(k+l+2)$-tuple encodes a pair of IDs, and so there is a quantifier-free first-order formula which ascertains whether a $6(k+l+2)$-tuple encodes three pairs of IDs (α_1, β_1), (α_2, β_2) and (α_3, β_3) such that (α_1, β_1) and (α_2, β_2) yield (α_3, β_3). Hence, by Proposition 31, Ω can be defined by a sentence of the form

$$\mathrm{PS}[\lambda \mathbf{x}, \mathbf{y}, \mathbf{z}, \psi(\mathbf{x}, \mathbf{y}, \mathbf{z})](\mathbf{0}, \mathbf{max}),$$

where $|\mathbf{x}| = |\mathbf{y}| = |\mathbf{z}| = 2(k+l+2)$, ψ is quantifier-free first-order and $\mathbf{0}$ (resp. $\mathbf{max}$) is the constant symbol $\mathbf{0}$ (resp. $\mathbf{max}$) repeated $2(k+l+2)$ times. Thus, in the presence of two built-in constants, $\mathrm{PS}(1) = \mathrm{NPSS}(1)$.

By proceeding by induction on m and using essentially the above construction, the main result follows. We remark that when we proceed as above for a program scheme of NPSS(3), say, and use the fact that $\mathrm{PS}(1) = \mathrm{NPSS}(1)$, we may need to replace a positive Boolean combination, i.e., just involving $\vee$ and $\wedge$, of atomic and negated atomic

formulae and formulae of the form (resp. $\neg$) $\forall w_1 \forall w_2 \ldots \forall w_p \mathrm{PS}[\lambda \mathbf{x}, \mathbf{y}, \mathbf{z} \psi](\mathbf{u}, \mathbf{v})$, with one formula of this form. That this can always be done is straightforward. Finally, by 'building our built-in constants from within', using existential quantification, we obtain the final parenthetic extension.

So, the class of problems NPSS has an equivalent formulation as the extension of first-order logic by the path system operator PS. It is known that in the presence of a built-in successor relation, $(\pm\mathrm{PS})^*[\mathrm{FO}] = \mathbf{P}$, and that any problem in $\mathbf{P}$ can be described by a sentence of $(\pm\mathrm{PS})^*[\mathrm{FO}]$ of the form

$$\mathrm{PS}[\lambda \mathbf{x}, \mathbf{y}, \mathbf{z} \psi(\mathbf{x}, \mathbf{y}, \mathbf{z})](\mathbf{0}, \mathbf{max}),$$

where ψ is quantifier-free first-order (involving the built-in successor relation) [46]. Consequently, we obtain the following result.

Theorem 33 In the presence of a built-in successor relation,

$$\mathrm{NPSS} = \mathrm{NPSS}(1) = (\pm\mathrm{PS})^*[\mathrm{FO}] = \mathrm{PS}^1[\mathrm{FO}] = \mathbf{P}.$$

Theorem 33 provides strong evidence that the problem PS is not in transitive closure logic, given Theorem 12 (or, equivalently, that PS is not in NPS, given Theorem 11); but as yet, we have not established whether this is true or not. In fact, it is the case that $\mathrm{PS} \notin (\pm\mathrm{TC})^*[\mathrm{FO}]$, and this fact can be established from existing results. The class of problems definable by the sentences of *existential fixed point logic* (that is, the fragment of $\mathrm{LFP}^*[\mathrm{FO}]$ defined by forbidding the universal quantifier and only allowing $\neg$ to negate atomic formulae) and the class of problems definable by the sentences of the existential fragment (defined in the same way) of $\mathrm{PS}^*[\mathrm{FO}]$ are one and the same [36]. Moreover, in the presence of two built-in constants, every problem in the existential fragment of $\mathrm{PS}^*[\mathrm{FO}]$ can be defined by a sentence of the form

$$\mathrm{PS}[\lambda \mathbf{x}, \mathbf{y}, \mathbf{z} \psi](\mathbf{0}, \mathbf{max}),$$

where ψ is quantifier-free first order [23, 36]. Also, there are problems in existential least fixed point logic that are not in transitive closure logic [23, 26] (in fact, problems involving rooted undirected trees). Hence, the problem PS is not in transitive closure logic. By Theorems 11 and 32, we obtain the following.

Theorem 34

$$(\pm\mathrm{TC})^*[\mathrm{FO}] = \mathrm{NPS} \subset \mathrm{NPSS} = (\pm\mathrm{PS})^*[\mathrm{FO}],$$

even when we only consider problems involving rooted undirected trees.

7 More hierarchy results

We now return to establishing a hierarchy theorem for the class of program schemes NPSS, analogous to that in NPS. We can, in fact, obtain such a theorem using a powerful existing result (obtained by playing games!) due to Grädel and McColm [22]. However, their result does not yield a hierarchy result holding on undirected trees or on out-trees: we need to consider computations in our program schemes of NPSS, as we did before for NPS, to obtain such a refined result. But first, we give Grädel and McColm's result and show how it can be applied.

Definition 35 Let w be a word over $\{\exists, \forall, T, N\}$, with ϵ the empty word. Then:

- $\mathrm{TC}(\epsilon)$ consists of all quantifier-free first-order formulae;
- for $Q \in \{\exists, \forall\}$, the class of formulae $\mathrm{TC}(Qw) \subseteq (\pm\mathrm{TC})^*[\mathrm{FO}]$ is the closure under conjunctions and disjunctions of $\mathrm{TC}(w) \cup \{(Qx_i)\varphi : \varphi \in \mathrm{TC}(w)\}$;
- the class of formulae $\mathrm{TC}(Tw) \subseteq (\pm\mathrm{TC})^*[\mathrm{FO}]$ is the closure under conjunctions and disjunctions of the class of formulae of the form $\mathrm{TC}[\lambda \mathbf{x}, \mathbf{y}\varphi](\mathbf{u}, \mathbf{v})$, where $\varphi \in \mathrm{TC}(w)$; and
- the class of formulae $\mathrm{TC}(Nw) \subseteq (\pm\mathrm{TC})^*[\mathrm{FO}]$ is the closure under conjunctions and disjunctions of the class of formulae of the form $\neg\mathrm{TC}[\lambda \mathbf{x}, \mathbf{y}\neg\varphi](\mathbf{u}, \mathbf{v})$, where $\varphi \in \mathrm{TC}(w)$.

Clearly, $\cup\{\mathrm{TC}(w) : w \in \{\exists, \forall, T, N\}^*\} = (\pm\mathrm{TC})^*[\mathrm{FO}]$.

For any $w \in \{\exists, \forall, T, N\}^*$, let $\tilde{w}$ be the word over $\{\exists, \forall, \exists^*, \forall^*\}$ obtained by replacing T by $\exists^*$ and N by $\forall^*$. Such a word $\tilde{w}$ also denotes the set of words obtained from $\tilde{w}$ by replacing any occurrence of $\exists^*$ (resp. $\forall^*$) with any word from $\{\exists\}^*$ (resp. $\{\forall\}^*$); that is, $\tilde{w}$ also denotes the set of words over $\{\exists, \forall\}$ denoted by the 'regular expression' $\tilde{w}$. For any $w \in \{\exists, \forall\}^*$, let $\bar{w}$ be obtained from w by replacing every $\exists$ with $\forall$ and vice versa.

Definition 36 The logic $\mathcal{L}_{\infty\omega}$ is formed using the usual operations of first-order logic except that conjunctions and disjunctions of arbitrary, not just finite, sets of formulae are allowed. The fragment $\mathcal{L}^k_{\infty\omega}$ consists of all formulae of $\mathcal{L}_{\infty\omega}$ in which at most k distinct variables appear; and *bounded variable infinitary logic* $\mathcal{L}^\omega_{\infty\omega}$ is defined as $\{\psi \in \mathcal{L}^k_{\infty\omega} : k \geq 0\}$.

Definition 37 Every formula ψ of bounded variable infinitary logic $\mathcal{L}^\omega_{\infty\omega}$ has a certain *quantifier structure* $P(\psi) \subseteq \{\exists, \forall\}^*$, defined as follows:

- if ψ is quantifier-free then $P(\psi) = \{\epsilon\}$;
- if ψ is of the form $\neg\varphi$ then $P(\psi) = \{\bar{w} : w \in P(\varphi)\}$;
- if ψ is of the form $\exists x\varphi$ then $P(\psi) = \{\exists w : w \in P(\varphi)\}$, and similarly when ψ is of the form $\forall x\varphi$ then $P(\psi) = \{\forall w : w \in P(\varphi)\}$; and
- if ψ is of the form $\vee\{\varphi_i : i \in I\}$ or $\wedge\{\varphi_i : i \in I\}$, for some index set I, then $P(\psi) = \cup\{P(\varphi_i) : i \in I\}$.

For each $k \geq 1$, a set $P \subseteq \{\exists, \forall\}^*$ yields the class of formulae $\mathcal{L}^k_{\infty\omega}(P)$ defined as

$$\{\psi \in \mathcal{L}^k_{\infty\omega} \quad : \quad \text{for every } w \in P(\psi) \text{ there exists a word } w' \in P \text{ such that } w \text{ can be obtained by deleting some of the symbols of } w'\},$$

with $\mathcal{L}^\omega_{\infty\omega}(P) = \{\psi \in \mathcal{L}^k_{\infty\omega}(P) : k \geq 0\}$. Let P_i consist of all those strings of $\{\exists, \forall\}^*$ in which there are exactly i $\forall$ symbols. Then the set of *infinitary formulae with bounded number of universal quantifiers*, $\mathcal{L}^\omega_{\infty\omega}(\mathrm{BU})$, is defined as $\{\psi \in \mathcal{L}^\omega_{\infty\omega}(P_i) : i \geq 0\}$.

Now for Grädel and McColm's result.

Theorem 38 [22]

(a) The problem consisting of all those structures over σ_{2++} for which the vertex D is not reachable from the vertex C via a path in the undirected graph whose edge relation is given by E is not definable in $\mathcal{L}^{\omega}_{\infty\omega}(\mathrm{BU})$.

(b) Let w be obtained from a word of $\{\exists, \forall, T, N\}^*$ by continually replacing $\exists T$, $T\exists$ and TT by T, and $\forall N$, $N\forall$ and NN by N until no more reductions can be made, i.e., w is reduced; and let $w' \in \{\exists, \forall, \exists^*, \forall^*\}^*$. Then:

> the class of problems $\mathrm{TC}(w)$ contains a formula ψ_w which is equivalent to a formula of $\mathcal{L}^{\omega}_{\infty\omega}(w')$

if, and only if,

> every word in $\tilde{w}$ can be obtained from some word of w' by deleting some symbols.

Now we can apply Theorem 38. From above, the problem PS is in existential least fixed point logic, which in turn is a fragment of existential bounded variable infinitary logic (defined from $\mathcal{L}^{\omega}_{\infty\omega}$ as was existential least fixed point logic from LFP*[FO]: see [23]). For $m \geq 1$, let w'_m be a word over $\{\exists, \forall, \exists^*, \forall^*\}$ denoting the set of words over $\{\exists, \forall\}$ consisting of at most $m-1$ alternations of blocks of $\forall$ and $\exists$ and whose first symbol is $\exists$, if m is odd, and $\forall$ if m is even. A simple induction yields that every problem in $\pm\mathrm{PS}(m)$, for $m \geq 1$, can be defined by a sentence of $\mathcal{L}^{\omega}_{\infty\omega}(w'_m)$. For each $m \geq 1$, let w_m be the string $\exists\forall\exists\forall\ldots\exists$ of length m, if m is odd, and let w_m be the string $\forall\exists\forall\exists\ldots\exists$ of length m, if m is even. Theorem 38 yields that $\mathrm{TC}(w_{m+1}) \not\subseteq \mathcal{L}^{\omega}_{\infty\omega}(w'_m)$, for any $m \geq 1$. But $\mathrm{TC}(w_{m+1}) \subseteq \mathrm{PS}(m+1)$, for $m \geq 1$, and so, by Theorem 32, we obtain the following result.

Corollary 39 In the presence of two built-in constants,

$$\mathrm{NPSS}(1) \subset \ldots \subset \mathrm{NPSS}(m) \subset \mathrm{NPSS}(m+1) \subset \ldots \ldots \subset \cup\{\mathrm{NPSS}(m) : m \geq 1\} = \mathrm{NPSS},$$

$$\pm\mathrm{PS}(1) \subset \ldots \subset \pm\mathrm{PS}(m) \subset \pm\mathrm{PS}(m+1) \subset \ldots \ldots \subset \cup\{\pm\mathrm{PS}(m) : m \geq 1\} = (\pm\mathrm{PS})^*[\mathrm{FO}]$$

and

$$\mathrm{PS}(1) \subset \ldots \subset \mathrm{PS}(m) \subset \mathrm{PS}(m+1) \subset \ldots \ldots \subset \cup\{\mathrm{PS}(m) : m \geq 1\} = \mathrm{PS}^*[\mathrm{FO}].$$

Moreover, for every $m \geq 1$, there is a problem of Σ_{m+1}, if m is even, and Π_m, if m is odd, in the $(m+1)$th level of any of these hierarchies which is not definable in the mth level.

We have two remarks, one negative, one positive. First, from [22], Corollary 39 holds when we consider problems over a fixed signature but only when this signature

contains 3 binary relation symbols and 2 constant symbols: Theorem 38 can not be used to show that Corollary 39 holds on graphs or digraphs. Second, Theorem 38 does clearly suffice to show that the hierarchy within path system logic obtained by interleaving applications of the operator PS and negations is proper, and that the problem co-PS is not definable in PS*[FO] (we leave these applications as exercises).

Given the drawback as regards Corollary 39, we now seek to improve it. Ideally, we would like Theorem 13 to hold for the program schemes of NPSS. Let us look at the proof of this theorem for program schemes of NPS(1) and see what happens when we have a stack present. Adopting the nomenclature of Theorem 13, clearly we still have that $\tilde{\mathcal{A}}_k^{1+r} \models \rho \Rightarrow \tilde{\mathcal{B}}_k^{1+r} \models \rho$ as we simply 'mirror' any computation of ρ on $\tilde{\mathcal{A}}_k^{1+r}$ by a computation of ρ on $\tilde{\mathcal{B}}_k^{1+r}$. Suppose that $\tilde{\mathcal{B}}_k^{1+r} \models \rho$. The crux of the proof of Theorem 13 is that in an accepting computation of ρ on $\tilde{\mathcal{B}}_k^{1+r}$ we can 'avoid' the left-most copy of $\mathcal{B}^0$ on layer 0 of $\tilde{\mathcal{B}}_k^{1+r}$; and so obtain an accepting computation of ρ on $\tilde{\mathcal{A}}_k^{1+r}$. The question is: 'Can we do likewise in the presence of a stack?'. On the face of it, the answer is 'no'. Simply proceeding as we do in Theorem 13 might leave the two stacks (in the two different computations of ρ on $\tilde{\mathcal{A}}_k^{1+r}$ and $\tilde{\mathcal{B}}_k^{1+r}$) consisting of different elements; and so we lose the property that our two computations proceed in tandem, so to speak. However, whilst we can not apply the proof of Theorem 13 exactly in the presence of a stack, we can use certain properties of accepting computations of program schemes of NPSS(1) to achieve a result very similar to Theorem 13.

Adopt the assumptions of the statement of Theorem 13 except assume ρ to be in NPSS(m) and not NPS(m) and consider ρ on structures $\tilde{\mathcal{B}}_{3k+2}^{m+r}$ and $\tilde{\mathcal{A}}_{3k+2}^{m+r}$. With regard to Definitions 28, 29 and 30, note that they are only given for program schemes of NPSS(1). However, we can define the notions in these definitions relative to a program scheme $\rho \in$ NPSS(m), for any odd $m > 1$, simply by taking a 'top-level' view of ρ (as we described immediately prior to Lemma 14). Consequently, Proposition 31 holds when $\rho \in$ NPSS(m), for any odd $m \geq 1$ (as the original proof works for the general case).

Lemma 40 Set $m \geq 3$ to be odd and fix 0 and max as the elements u and v of $\tilde{\mathcal{A}}_{3k+2}^{m+r}$ such that $U_{m+r}(u)$ holds, $U_{m+r-1}(v)$ holds and v is in the right-most copy of $\mathcal{A}_{3k+2}^{m+r-1}$ or $\mathcal{B}_{3k+2}^{m+r-1}$ on layer $m+r-1$ of $\tilde{\mathcal{A}}_{3k+2}^{m+r}$. Let (α, β) be a realizable pair of IDs of ρ on input $\tilde{\mathcal{A}}_{3k+2}^{m+r}$ such that no value of α or β lies in the left-most copy of $\mathcal{A}_{3k+2}^{m-2}$ on layer $m-2$ of $\tilde{\mathcal{A}}_{3k+2}^{m+r}$. Then there is a computation of ρ on input $\tilde{\mathcal{A}}_{3k+2}^{m+r}$ from configuration $\bar{\alpha}$ to configuration $\bar{\beta}$ such that throughout this computation:

- no input-output variable ever takes a value from the left-most copy of $\mathcal{A}_{3k+2}^{m-2}$ on layer $m-2$ of $\tilde{\mathcal{A}}_{3k+2}^{m+r}$; and
- the height of the stack does not decrease.

Proof By Proposition 31, (α, β) can be obtained from the set Γ of all pairs of IDs of the form (γ, γ) by applying the yield rules. Hence, we proceed by induction on the number t of yield rules applied to obtain (α, β). Note that a simple induction yields that any pair of IDs obtained from Γ by applying the yield rules is realizable.

The base case of the induction is when $t = 1$. There are two ways in which (α, β) could have been obtained: via rule (a) or via rule (b) of Definition 29. If rule (a) was

applied then there is a computation of length 2 from configuration $\bar{\alpha}$ to configuration $\bar{\beta}$ which consists of a push followed by a pop. If rule (b) was applied then there is a computation of length 1 from configuration $\bar{\alpha}$ to configuration $\bar{\beta}$ where the move is neither a pop nor a push. In either case, the computation is as required.

Suppose, as our induction hypothesis, that the result holds for all realizable pairs of IDs that can be obtained from Γ by applying less than t yield rules. Let (α, β) be a realizable pair of IDs that can be obtained from Γ by applying t yield rules. Again, there are two ways in which (α, β) could have been obtained: via rule (a) or via rule (b) of Definition 29.

Suppose that (α_1, β_1) and (α_2, β_2) yield (α, β) by applying rule (a), where (α_1, β_1) and (α_2, β_2) can be obtained from Γ by applying less than t yield rules. In order to immediately apply the induction hypothesis, we need that no value of β_1, α_2 or β_2 lies in the left-most copy of $\mathcal{A}^{m-2}_{3k+2}$ on layer $m-2$ of $\tilde{\mathcal{A}}^{m+r}_{3k+2}$: however, this may not be the case. Consider the number of different values from amongst the IDs α_1, β_1, α_2, β_2 and β. There are: at most $k+1$ different values of α_1; at most another k different values of β_1 (note that the stack elements of α_1 and β_1, if they exist, are identical); every value of α_2 has already been accounted for (as a value of β_1); at most another k different values of β_2 (note that the stack elements of α_2 and β_2 are identical); and every value of β has already been accounted for (as a value of β_2). Hence, the set of different values, V, say, from the IDs α_1, β_1, α_2, β_2 and β has size at most $3k+1$.

If some of the values from V lie in the left-most copy of $\mathcal{A}^{m-2}_{3k+2}$ on layer $m-2$ of $\tilde{\mathcal{A}}^{m+r}_{3k+2}$ then let θ be the automorphism of $\tilde{\mathcal{A}}^{m+r}_{3k+2}$ obtained:

- by mapping every element in the left-most copy of $\mathcal{A}^{m-2}_{3k+2}$ on layer $m-2$ to its corresponding element in a 'free copy' of $\mathcal{A}^{m-2}_{3k+2}$ on layer $m-2$ from the $3k+2$ copies adjacent to the left-most copy, and vice versa; and
- by fixing every other element

(note that such a 'free copy' exists). If no value from V lies in the left-most copy of $\mathcal{A}^{m-2}_{3k+2}$ on layer $m-2$ of $\tilde{\mathcal{A}}^{m+r}_{3k+2}$ then let θ be the identity automorphism.

As (α_1, β_1) is realizable, there is a computation of ρ on input $\tilde{\mathcal{A}}^{m+r}_{3k+2}$ from configuration $\bar{\alpha}_1$ to configuration $\bar{\beta}_1$ such that throughout the computation, the stack height does not decrease. By mirroring this computation using the automorphism θ, the pair of IDs $(\alpha_1, \theta(\beta_1))$ is realizable (note that $\theta(\alpha_1) = \alpha_1$); and similarly, the pair of IDs $(\theta(\alpha_2), \theta(\beta_2))$ is realizable. Also, $(\alpha_1, \theta(\beta_1))$ and $(\theta(\alpha_2), \theta(\beta_2))$ can be obtained from Γ by applying less than t yield rules (simply use θ to mirror the yield rules used to obtain (α_1, β_1) and (α_2, β_2)). Finally, $(\alpha_1, \theta(\beta_1))$ and $(\theta(\alpha_2), \theta(\beta_2))$ yield (α, β) (as $\theta(\beta) = \beta$). Consequently, applying the induction hypothesis yields that there is a computation of ρ on input $\tilde{\mathcal{A}}^{m+r}_{3k+2}$ from configuration $\bar{\alpha}$ to configuration $\bar{\beta}$ such that throughout this computation: no input-output variable ever takes a value from the left-most copy of $\mathcal{A}^{m-2}_{3k+2}$ on layer $m-2$ of $\tilde{\mathcal{A}}^{m+r}_{3k+2}$; and the height of the stack does not decrease (note that all stack elements of the IDs α_1, β_1, α_2, β_2 and β are fixed by θ).

Suppose that (α_1, β_1) and (α_2, β_2) yield (α, β) by applying rule (b), where (α_1, β_1) and (α_2, β_2) can be obtained from Γ by applying less than t yield rules. A simple count yields that the set of different values from the IDs α_1, β_1, α_2, β_2 and β has size at most $3k+2$. Proceeding similarly to above gives the result.

Lemma 41 Set $m \geq 1$ to be odd and fix 0 and max as the elements u and v of $\bar{\mathcal{B}}^{m+r}_{3k+3}$ such that $U_{m+r}(u)$ holds, $U_{m+r-1}(v)$ holds and v is in the right-most copy of $\mathcal{A}^{m+r-1}_{3k+3}$ or $\mathcal{B}^{m+r-1}_{3k+3}$ on layer $m+r-1$ of $\tilde{\mathcal{B}}^{m+r}_{3k+3}$. Let (α, β) be a realizable pair of IDs of ρ on input $\bar{\mathcal{B}}^{m+r}_{3k+3}$ such that no value of α or β lies in the left-most copy of $\mathcal{B}^{m-1}_{3k+3}$ on layer $m-1$ of $\bar{\mathcal{B}}^{m+r}_{3k+3}$. Then there is a computation of ρ on input $\bar{\mathcal{B}}^{m+r}_{3k+3}$ from configuration $\bar{\alpha}$ to configuration $\bar{\beta}$ such that throughout this computation:

- no input-output variable ever takes a value from the left-most copy of $\mathcal{B}^{m-1}_{3k+3}$ on layer $m-1$ of $\tilde{\mathcal{B}}^{m+r}_{3k+3}$; and
- the height of the stack does not decrease.

Proof The result follows by proceeding similarly to the proof of Lemma 40. The only additional remark to make is that we need $\bar{\mathcal{B}}^{m+r}_{3k+3}$ in the statement of the lemma, as opposed to $\bar{\mathcal{B}}^{m+r}_{3k+2}$, because the constant max might interfere when $m = 1$.

Armed with Lemmas 40 and 41, we can obtain the following result.

Theorem 42 Let σ be some relational signature containing the unary relation symbol U_0 and let $\mathcal{A}^0$ and $\mathcal{B}^0$ be σ-structures such that:

- $\mathcal{A}^0 \subseteq \mathcal{B}^0$; and
- $|\{u \in |\mathcal{A}^0| : U_0(u) \text{ holds in } \mathcal{A}^0\}| = |\{u \in |\mathcal{B}^0| : U_0(u) \text{ holds in } \mathcal{B}^0\}| = 1$.

Fix $m \geq 1$, $k \geq 1$ and $r \geq 0$, and:

- let $\rho \in \text{NPSS}(m)$ be over the signature σ^{m+r} and involve k variables, s of which are free; and
- let $\tilde{\mathcal{A}}^{m+r}_{3k+3}$ and $\tilde{\mathcal{B}}^{m+r}_{3k+3}$ be expansions of the σ^{m+r}-structures $\mathcal{A}^{m+r}_{3k+3}$ and $\mathcal{B}^{m+r}_{3k+3}$ by adjoining s constants (one for each free variable of ρ) so that:
  - $\tilde{\mathcal{A}}^{m+r}_{3k+3} \subseteq \tilde{\mathcal{B}}^{m+r}_{3k+3}$ via a natural embedding π which embeds the left-most copy of $\mathcal{A}^{m-1}_{3k+3}$ on layer $m-1$ of $\tilde{\mathcal{A}}^{m+r}_{3k+3}$ into the left-most copy of $\mathcal{B}^{m-1}_{3k+3}$ on layer $m-1$ of $\tilde{\mathcal{B}}^{m+r}_{3k+3}$; but
  - none of the adjoined constants lie in the left-most copy of $\mathcal{A}^{m-1}_{3k+3}$ on layer $m-1$ of $\tilde{\mathcal{A}}^{m+r}_{3k+3}$ nor in the left-most copy of $\mathcal{B}^{m-1}_{3k+3}$ on layer $m-1$ of $\tilde{\mathcal{B}}^{m+r}_{3k+3}$.

Then

$$\tilde{\mathcal{A}}^{m+r}_{3k+3} \models \rho \text{ if, and only if, } \tilde{\mathcal{B}}^{m+r}_{3k+3} \models \rho.$$

Proof The proof of Theorem 13 goes through with Lemmas 40 and 41 playing the roles of Lemmas 14 and 15. Our only additional comment is that, by Remark 26, $\tilde{\mathcal{A}}^{m+r}_{3k+3} \models \rho$ if, and only if, (α_0, β_0) is realizable, where α_0 and β_0 are the unique initial and accepting IDs, respectively (there is an analogous statement concerning $\tilde{\mathcal{B}}^{m+r}_{3k+3}$).

We can now obtain our basic hierarchy theorem for NPSS. The proof of this result proceeds exactly as do those of Corollaries 17 and 18 except that we use Theorem 42 in place of Theorem 13.

Corollary 43 If $m \geq 2$ is even then there are problems in Π_m which are not in NPSS$(m-1)$, and if $m \geq 3$ is odd then there are problems in Σ_m which are not in NPSS$(m-1)$. In particular,

$$\text{NPSS}(1) \subset \ldots \subset \text{NPSS}(m) \subset \text{NPSS}(m+1) \subset \ldots$$

Moreover, the above results hold even when we only consider problems involving undirected trees or problems involving out-trees.

Also, Theorem 32 and Corollary 43 yield the following result.

Corollary 44 In the presence of two built-in constants, if $m \geq 2$ is even then there are problems in Π_m which are not in $\pm\text{PS}(m-1)$, and if $m \geq 3$ is odd then there are problems in Σ_m which are not in $\pm\text{PS}(m-1)$. In particular,

$$\text{PS}(1) \subset \ldots \subset \text{PS}(m) \subset \text{PS}(m+1) \subset \ldots \\ \ldots \subset \cup\{\text{PS}(m) : m \geq 1\} = \text{PS}^*[\text{FO}]$$

and

$$\pm\text{PS}(1) \subset \ldots \subset \pm\text{PS}(m) \subset \pm\text{PS}(m+1) \subset \ldots \\ \ldots \subset \cup\{\pm\text{PS}(m) : m \geq 1\} = (\pm\text{PS})^*[\text{FO}]$$

Moreover, the above results hold even when we only consider problems involving undirected trees or problems involving out-trees.

8 Conclusion

We begin our conclusion by returning to the question of whether there is a logic for **P**, as mentioned in the Introduction. As we said there, our purpose in considering the program schemes of NPS and NPSS here is not really to try and concoct some class of program schemes (without built-in relations) to capture **P** or to increase the class of problems captured in comparison with other previously proposed logical characterizations of **P**. It is to: first, examine the classes of problems NPS and NPSS as problem classes in their own right, given that the two formalizations are, to our minds, quite natural (recall Cook's result [9] that non-deterministic pushdown automata recognize exactly the polynomial-time recognizable languages over $\{0,1\}$); and, second, to look for equivalent, logical characterizations of NPS and NPSS, and apply these characterizations to obtain new logical results. We feel we have been quite successful in this regard, especially given that all our results have been obtained without recourse to Ehrenfeucht-Fraïssé games. However, we are also conscious of the fact that we should investigate the relationship between NPSS and some other previously proposed logical characterizations of **P**. We do that here, and show that there are problems in both $\text{LFP}^1[\text{FO}]$ and $\text{ATC}^1[\text{FO}]$ (which are fragments of least fixed point logic and alternating transitive closure logic, respectively [13]) which are not definable in NPSS. We use the fact that $\text{LFP}^1[\text{FO}] = \text{ATC}^1[\text{FO}]$ (see [13, Theorem 8.4.8]) and only exhibit a problem from $\text{LFP}^1[\text{FO}]$ that is not in NPSS.

Proposition 45 Adopt the nomenclature of Proposition 16. For every vertex x of $\mathcal{A}_k^m$ and $\mathcal{B}_k^m$ for which $U_i(x)$ holds, for $i < m$ even, place a new vertex on the edge joining x with the vertex y for which $U_{i+1}(y)$ holds. Now regard (these amended) $\mathcal{A}_k^m$ and $\mathcal{B}_k^m$ simply as out-trees with edges directed away from the vertex z for which $U_m(z)$ formerly held (i.e., the reducts of the original structures to σ_2). Then there are sentences Φ_e and Φ_o of $\mathrm{LFP}^1|\mathrm{FO}(\sigma_2)|$ such that for any $k \geq 1$,

$$\mathcal{A}_k^m \models \Phi_e \text{ and } \mathcal{B}_k^m \not\models \Phi_e \text{ when } m \text{ is even}$$

and

$$\mathcal{A}_k^m \models \Phi_o \text{ and } \mathcal{B}_k^m \not\models \Phi_o \text{ when } m \text{ is odd.}$$

Proof Define the following predicates:

- $d_o^0(x)$ to be $\forall y \neg E(x,y)$ ('x has out-degree 0');
- $d_i^0(x)$ to be $\forall y \neg E(y,x)$ ('x has in-degree 0'); and
- $d_o^1(x)$ to be $\exists y(E(x,y) \wedge \forall z(E(x,z) \Rightarrow y = z))$ ('x has out-degree 1').

Let R be a new relation symbol of arity 1. If $m \geq 2$ is even then define φ_e as

$$\begin{aligned} d_o^0(x) \quad \vee \quad & (\exists y(d_o^1(y) \wedge E(y,x)) \wedge \forall z(E(x,z) \Rightarrow R(z))) \\ \vee \quad & (\neg d_i^0(x) \wedge \forall y(d_o^1(y) \Rightarrow \neg E(y,x)) \wedge \forall y(E(x,y) \Rightarrow \neg d_o^0(y)) \\ & \wedge \exists z(E(x,z) \wedge R(z))) \\ \vee \quad & (d_i^0(x) \wedge \forall y(E(x,y) \Rightarrow R(y))) \end{aligned}$$

and if $m \geq 1$ is odd then define φ_o as

$$\begin{aligned} d_o^0(x) \quad \vee \quad & (\exists y(d_o^1(y) \wedge E(y,x)) \wedge \forall z(E(x,z) \Rightarrow R(z))) \\ \vee \quad & (\neg d_i^0(x) \wedge \forall y(d_o(1)(y) \Rightarrow \neg E(y,x)) \wedge \forall y(E(x,y) \Rightarrow \neg d_o^0(y)) \\ & \wedge \exists z(E(x,z) \wedge R(z))) \\ \vee \quad & (d_i^0(x) \wedge \exists y(E(x,y) \wedge R(y))). \end{aligned}$$

Essentially, the first lines of φ_e and φ_o set the leaves of the out-trees $\mathcal{A}_k^m$ and $\mathcal{B}_k^m$ to be in the relation R and give the rules for adding a vertex x to R when x was formerly a vertex for which $U_i(x)$ held, for some even $i < m$. The second lines give the rules for adding a vertex x for which $U_i(x)$ held, for some odd $i < m$, to R and also the rules for adding the 'new' vertices of $\mathcal{A}_k^m$ and $\mathcal{B}_k^m$ to R. Finally, the final lines give the rules for adding the root to R.

If we define the sentences Φ_e and Φ_o as

$$\forall x(d_i^0(x) \Rightarrow \mathrm{LFP}[\lambda x, R, \varphi_e(x)](x))$$

and

$$\forall x(d_i^0(x) \Rightarrow \mathrm{LFP}[\lambda x, R, \varphi_o(x)](x)),$$

respectively, the result follows.

With regard to the discussion following Corollary 17, we could easily prove a similar result to Proposition 45 for undirected trees $\mathcal{A}_k^m$ and $\mathcal{B}_k^m$. By applying Theorems 32 and 42, the following is immediate.

Corollary 46

$$\text{NPSS} = (\pm\text{PS})^*[\text{FO}] \subset \text{LFP}^1[\text{FO}] = \text{ATC}^1[\text{FO}],$$

even when we only consider problems involving undirected trees or problems involving out-trees.

Let us draw to a close by pulling together the contributions in this paper. We have developed an alternative to defining classes of problems using logic by considering program schemes; which are, essentially, high-level models of computation taking finite structures as their inputs. We have shown that the class of problems accepted by the program schemes of NPS coincides with the class of problems definable by the sentences of transitive closure logic, and we have used this identification to exhibit proper infinite hierarchies within transitive closure logic. Importantly, we did this without recourse to any sort of Ehrenfeucht-Fraïssé games (the tools previously used to establish many hierarchy results), and we simply considered computations of our program schemes on specific finite structures.

Our consideration of computational devices, as opposed to logical formulae, enabled us to increase the power of the program schemes of NPS by adding in a stack; an option not really available in the logical setting. We showed that the class of problems accepted by the program schemes of NPSS has an equivalent formulation as the class of problems defined by the sentences of path system logic: this characterization was not previously known. Furthermore, we established the (hitherto unknown) fact that there are proper infinite hierarchies within path system logic. Again, our logical hierarchy results for path system logic have been established without playing any sort of Ehrenfeucht-Fraïssé game. We feel that the general approach of equating classes of problems accepted by appropriate computational devices with those defined by the formulae of logics has a rosy future; and we hope that such characterizations will yield new logical inexpressibility results, obtained by considering computations as opposed to playing games.

Finally, we mention some directions for further research. We would like to consider adding other high-level programming language constructs, such as an array or arrays, to the program schemes of NPS and NPSS. It is to be hoped that doing so might yield proper infinite hierarchies within a logic $(\pm\Omega)^*[\text{FO}]$ where Ω is, for example, an operator corresponding to a **PSPACE**-complete problem (the only result known in this context regarding the expressive power of a logic formed by extending first-order logic with an operator corresponding to a **PSPACE**-complete problems is a minor inexpressibility result in [3]). We would also like to consider adding new constructs to NPSS so as to increase computing power yet stay within **P**.

All of our hierarchy results hold over the signature σ_2. This leaves open the status of these results when we restrict our signatures to only contain unary relation symbols. Grädel and McColm [22] remark that over signatures containing only unary relation symbols, all first-order formulae are logically equivalent to formulae in Δ_2, i.e., $\Sigma_2 \cap \Pi_2$. This result may be of some help.

We have shown how Grädel and McColm's main result in [22] can be applied to yield basic hierarchy results in NPS, transitive closure logic, NPSS and path system logic, and we have also highlighted its deficiencies with regard to our approach. Nevertheless, the result of Grädel and McColm's is very powerful and it is worth pursuing as to whether their result can be extended so that it can be applied to undirected graphs or digraphs (recall, at present it can only be applied to signatures containing 3 binary relation symbols and 2 constant symbols).

Acknowledgement. We are indebted to two referees for their very careful readings of draft versions of this paper which resulted in a number of errors being discovered and a number of significant omissions from the general literature being included. Their diligence and perseverence has improved this paper significantly.

References

[1] S. Abiteboul and V. Vianu. Fixpoint extensions of first-order logic and Datalog-like languages. In *Proc. 4th Ann. IEEE Symp. on Logic in Computer Science*, pages 71–79. IEEE Press, 1989.

[2] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proc. 23rd Ann. ACM Symp. on Theory of Computing*, pages 209–219. ACM Press, 1991.

[3] A.A. Arratia-Quesada and I.A. Stewart. Generalized Hex and logical characterizations of polynomial space. *Inform. Process. Lett.*, 63:147–152, 1997.

[4] D.A.M. Barrington, N. Immerman, and H. Straubing. On uniformity within NC^1. *J. Comput. System Sci.*, 41:274–306, 1990.

[5] J. Barwise. On Moschovakis closure ordinals. *J. Symbolic Logic*, 42:292–296, 1977.

[6] S. Brown, D. Gries, and T. Szymanski. Universality of data retrieval languages. In *Proc. 6th Ann. ACM Symp. on Principles of Programming Languages*, pages 110–117, 1979.

[7] A. Chandra and D. Harel. Structure and complexity of relational queries. *J. Comput. System Sci.*, 25:99–128, 1982.

[8] R. Constable and D. Gries. On classes of program schemata. *SIAM J. Comput.*, 1:66–118, 1972.

[9] S.A. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *J. Assoc. Comput. Mach.*, 18:4–18, 1971.

[10] S.A. Cook. An observation on time-storage trade off. *J. Comput. System Sci.*, 9:308–316, 1974.

[11] B. Courcelle. Recursive applicative program schemes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Vol. B*, pages 459–492. Elsevier, 1990.

[12] E. Dahlhaus. Reductions to NP complete problems by interpretations. In *Lecture Notes in Computer Science Vol. 171*, pages 357–365. Springer-Verlag, 1984.

[13] H.D. Ebbinghaus and J. Flum. *Finite Model Theory.* Springer-Verlag, 1995.

[14] A. Ehrenfeucht. An application of games to the completeness problem for formalized theories. *Fund. Math.*, 49:129–141, 1961.

[15] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In R.M. Karp, editor, *Complexity of Computation*, volume 7 of *SIAM-AMS Proceedings*, pages 43–73, 1974.

[16] R. Fagin. Monadic generalized spectra. *Zeitschrift f. Math. Logik Grundlagen d. Math.*, 21:89–96, 1975.

[17] R. Fraïssé. Sur quelques classifications des systèms de relations. *Publ. Sci. Univ. Alger.*, 1:35–182, 1954.

[18] H. Friedman. Algorithmic procedures, generalized Turing algorithms and elementary recursion theory. In R.O. Gandy and C.M.E. Yates, editors, *Logic Colloquium 1969*, pages 361–390. North-Holland, 1971.

[19] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[20] G. Gottlob. Relativized logspace and generalized quantifiers over finite ordered structures. *J. Symbolic Logic*, 62:545–574, 1997.

[21] E. Grädel. On transitive closure logic. In *Lecture Notes in Computer Science Vol. 626*, pages 149–163. Springer-Verlag, 1992.

[22] E. Grädel and G. McColm. Hierarchies in transitive closure logic, stratified datalog and infinitary logic. *Ann. Pure App. Logic*, 77:166–199, 1996.

[23] M. Grohe. Existential least fixed-point logic and its relatives. *J. Logic Computat.*, 7:205–228, 1997.

[24] D. Harel and D. Peleg. On static logics, dynamic logics, and complexity classes. *Inform. Control*, 60:86–102, 1984.

[25] W. Hodges. *Model Theory.* Cambridge University Press, 1993.

[26] N. Immerman. Number of quantifiers is better than number of tape cells. *J. Comput. System Sci.*, 22:65–72, 1981.

[27] N. Immerman. Upper and lower bounds for first-order expressibility. *J. Comput. System Sci.*, 25:76–98, 1982.

[28] N. Immerman. Relational queries computable in polynomial time. *Inform. Control*, 68:86–104, 1986.

[29] N. Immerman. Languages that capture complexity classes. *SIAM J. Comput.*, 16:760–778, 1987.

[30] N. Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17:935–938, 1988.

[31] N.D. Jones and W.T. Laaser. Complete problems for deterministic polynomial time. *Theoret. Comput. Sci.*, 3:105–117, 1977.

[32] N.D. Jones and S.S. Muchnik. Even simple programs are hard to analyze. *J. Assoc. Comput. Mach.*, 24:338–350, 1972.

[33] Ph.G. Kolaitis and M.N. Thakur. Logical definability of NP-optimization problems. *Inform. Computat.*, 87:302–338, 1990.

[34] Ph.G. Kolaitis and J. Väänänen. Generalized quantifiers and pebble games on finite structures. *Ann. Pure App. Logic*, 74:23–75, 1995.

[35] D. Kozen and J. Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Vol. B*, pages 789–840. Elsevier, 1990.

[36] C. Lautemann, T. Schwentick, and I.A. Stewart. Positive versions of polynomial time. *Inform. Computat.*, 147:145–170, 1998.

[37] D. Leivant. Descriptive characterizations of computational complexity. *J. Comput. System Sci.*, 39:51–83, 1989.

[38] G. McColm. Pebble games and subroutines in least fixed point logic. *Inform. Computat.*, 122:201–220, 1995.

[39] M. Otto. *Bounded variable logics and counting.* Lecture Notes in Logic Vol. 9. Springer-Verlag, 1997.

[40] M. Paterson and N. Hewitt. Comparative schematology. In *Record of Project MAC Conf. on Concurrent Systems and Parallel Computation*, pages 119–128. ACM Press, 1970.

[41] I.A. Stewart. Complete problems involving boolean labelled structures and projection translations. *J. Logic Computat.*, 1:861–882, 1991.

[42] I.A. Stewart. Using the Hamiltonian path operator to capture NP. *J. Comput. Systems Sci.*, 45:127–151, 1992.

[43] I.A. Stewart. Logical and schematic characterization of complexity classes. *Acta Informat.*, 30:61–87, 1993.

[44] I.A. Stewart. Logical characterizations of bounded query classes I: logspace oracle machines. *Fund. Informat.*, 18:65–92, 1993.

[45] I.A. Stewart. Logical characterizations of bounded query classes II: polynomial-time oracle machines. *Fund. Informat.*, 18:93–105, 1993.

[46] I.A. Stewart. Logical description of monotone NP problems. *J. Logic Computat.*, 4:337–357, 1994.

[47] J. Tiuryn and P. Urzyczyn. Some relationships between logics of programs and complexity theory. *Theoret. Comput. Sci.*, 60:83–108, 1988.

[48] B.A. Trakhtenbrot. Impossibility of an algorithm for the decision problem on finite classes. *Doklady*, 70:569–572, 1950.

[49] M. Vardi. Complexity of relational query languages. In *Proc. 14th ACM Ann. Symp. on Theory of Computing*, pages 137–146. ACM Press, 1982.